jPET: Un generador automático de casos de prueba sobre programas Java

Proyecto de Sistemas Informáticos

Memoria presentada por
Israel Cabañas Ruiz
Antonio Flores Montoya
y **Sergio Gutiérrez Mota**

Dirigido por los profesores
Elvira Albert Albiol
Miguel Gómez-Zamalloa Gil

Facultad de Informática
Universidad Complutense de Madrid
Madrid, Junio de 2011

Los abajo firmantes autorizan a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Fdo: Israel Cabañas Ruiz Fdo: Antonio Flores Montoya Fdo: Sergio Gutiérrez Mota

Resumen

PET es una herramienta ya existente para la *generación de casos de prueba* que recibe como entrada un *código de bytes* de Java y una selección entre los criterios de recubrimiento disponibles y obtiene como salida un conjunto de casos de prueba (*test-cases*) que garantizan el recubrimiento seleccionado. Puesto que el código de bytes es una representación de bajo nivel del programa, la información inferida por PET es difícilmente interpretable por el usuario. Esto en concreto impide la utilización de PET durante el proceso de desarrollo de software, área en la que el *testing* tiene un amplio campo de aplicación. El objetivo del presente proyecto es la extensión de PET para su uso sobre programas Java de alto nivel y su integración en el entorno de desarrollo integrado Eclipse, con el objetivo de poder usar los resultados obtenidos por PET durante el proceso de desarrollo de software.

El presente proyecto, al que hemos nombrado jPET, hace especial hincapié en el tratamiento posterior de la información generada por PET con el objetivo de presentarla al usuario de una forma fácil de entender. jPET incorpora un visor de casos de prueba (test-case viewer) que puede mostrar el contenido de la memoria antes de la ejecución (heap de entrada) y después de la ejecución de cada caso de prueba (heap de salida). jPET puede mostrar la traza de ejecución de un caso de prueba dado (i.e., la secuencia de instrucciones que el caso de prueba ejecutaría) de dos formas distintas: (1) marcando todas las instrucciones implicadas o (2) permitiendo al usuario reproducir la secuencia de instrucciones paso a paso usando la interfaz de depuración de Eclipse. Por último, puede analizar sintácticamente precondiciones de métodos escritas en JML (Java modeling language) y usarlas para evitar la generación de casos de prueba poco interesantes.

Las principales contribuciones del proyecto se han recogido en un artículo titulado *Software testing using jPET* [2] que tenemos previsto enviarlo próximamente a un congreso internacional.

Palabras clave: Testing, Generación de casos de prueba, Ejecución simbólica, Precondiciones, Plugin de Eclipse

Abstract

PET is an existing *test case generation tool* that takes as input a *Java bytecode* program and a selection of a coverage criteria (among those available in the system) and returns in the output a set of test-cases which ensure the selected coverage. As bytecode is a low-level representation of the program, the information obtained by PET is difficult to interpret by a non-expert user. This particularly prevents the use of PET during software development, an area in which *testing* has a large application field. The goal of this project is the extension of PET for its use on Java source programs and its integration within the Eclipse integrated development environment with the objective of being able to apply the results obtained by PET during sofware development.

This project, named jPET, puts special emphasis on advanced processing of the information generated by PET in order to display it to the user in an easy way to interpret. For this purpose, jPET incorporates a viewer of test cases (test-case viewer) that can display the contents of the memory before execution (heap entry) and after the execution of each test case (heap output). jPET can show the execution trace of a given test case (i.e., the sequence of instructions that execute the test case) in two ways: (1) by marking all instructions involved or (2) by allowing the user to reproduce the sequence of instructions step by step using the Eclipse debugger interface. Finally, jPET can parse preconditions of methods written in JML (Java Modeling Language) and use them to avoid the generation of test cases which are not interesting.

The main contributions of this project have been written in a paper entitled *Software testing using jPET* [2] that will be submitted soon to an Internacional Conference.

Keywords: Testing, Test-case generation, Symbolic execution, Preconditions, Eclipse plugin

Índice general

Capítulo 1

Introducción

1.1. Software Testing

El Software Testing es hoy en día una disciplina esencial de la ingeniería del software. Las fases de prueba han demostrado ser uno de los elementos más largos y costosos de los ciclos de desarrollo de software. Esto se debe a su dificultad y tiempo requerido. Pueden hacer falta una gran cantidad de pruebas para comprobar el funcionamiento de una aplicación y aun así no hay garantías de la inexistencia de errores.

Esta dificultad adquiere mayor relevancia a medida que la complejidad de los productos de software aumenta. Además, la realización de pruebas exhaustivas juega un papel principal en ámbitos relacionados con la seguridad o en el diseño de sistemas críticos. No sólo es necesario la realización de pruebas, sino que es conveniente comenzar a hacerlas en etapas tempranas del desarrollo. El coste asociado a los errores de programación es mayor cuanto más tarde es detectado en el proceso de desarrollo [9].

Existe por lo tanto la necesidad de herramientas que faciliten el desarrollo de tests y pruebas para un software en desarrollo dado. En concreto, la herramienta presentada genera automáticamente pruebas de software de una forma sencilla y rápida permitiendo la realización temprana y frecuente de pruebas durante el desarrollo de programas Java.

1.2. Conceptos básicos

1.2.1. Tipos de testing

En general las pruebas de software se suelen dividir en dos categorías principales, el *black-box testing* y el *white-box testing*. El black-box testing son pruebas que se realizan sin ningún conocimiento sobre la implementación o el funcionamiento interno del programa y basándose únicamente en los requisitos del software. El white-box testing, al contrario, se realiza teniendo en cuenta la implementación del programa, es decir, ejecutando el programa a testear.

Las pruebas de software suelen clasificarse también atendiendo a otros criterios como pueden ser el elemento de software probado ó la fase del desarrollo en que se llevan a cabo. Algunas categorías destacables son las siguientes:

- El *unit testing* tiene como objeto unidades o módulos individuales de código fuente. Clases o métodos en el caso de la programación orientada a objetos.

- El *integration testing* se realiza cuando se integran distintos módulos para que trabajen juntos.

- El *system testing* permite comprobar el funcionamiento de un sistema software completo.

- *Acceptance testing* es el conjunto de pruebas que realiza el cliente para validar el producto. En ocasiones se incluye como parte del contrato con el cliente.

- *Regression testing* se encarga de asegurar que todo el software sigue funcionando después de que se hayan realizado cambios. Consiste en una batería de pruebas que se ejecuta periódicamente y de forma automática.

El presente proyecto se encuadra en el contexto del white-box testing ya que la generación automática de pruebas se basa en el análisis del código fuente y se puede clasificar como unit testing al tener como objeto de análisis métodos individuales.

1.2.2. Casos de prueba y recubrimiento

Un caso de prueba, denominado en inglés *test-case*, es un conjunto de condiciones o valores iniciales de un programa o elemento software que determina la ejecución del mismo y permite al realizador de las pruebas comprobar el buen funcionamiento de dicho elemento software. Esta comprobación se realiza comparando el resultado esperado del programa, con el resultado real de la ejecución. La persona o sistema que suministra los resultados esperados se denomina *oráculo*.

En general no es posible especificar casos de prueba para explorar todas las posibilidades de un software. Estas posibilidades suelen ser intratables debido a su elevado número y en algunos casos infinitas. Por esta razón, se establece una medida para evaluar la capacidad de un conjunto de casos de prueba para ejercitar un software. Esta medida, llamada comúnmente recubrimiento, indica hasta que punto un elemento software ha sido probado.

Existen multitud de criterios de recubrimiento, los más comunes son:

- Recubrimiento de funciones: Mide si todas las funciones o subrutinas de un código fuente han sido probadas.
- Recubrimiento de instrucciones: Comprueba si todas las instrucciones del código analizado han sido ejecutadas.
- Recubrimiento de condiciones: Evalúa si todas las subexpresiones booleanas existentes en un programa o parte de él han sido evaluadas a cierto y a falso.
- Recubrimiento *Loop-k* : Asegura que se han ejecutado k iteraciones, o el máximo número de iteraciones posibles si este máximo es menor que k, en todos los bucles del programa analizado.

1.3. Derivación de casos de prueba

Dado un elemento software, es posible derivar casos de prueba del código fuente basándose en un criterio de recubrimiento. Este proceso busca obtener configuraciones de los datos de entrada que ejerciten distintas partes del

código, basándose en el propio código. Explicado de forma sencilla, el criterio de recubrimiento indica que puntos del código se quieren alcanzar y el proceso de derivación de casos de prueba consiste en hallar las condiciones que tiene que cumplir la entrada para llegar a dichos puntos. El criterio de recubrimiento guía el proceso de derivación. Finalmente, los sistemas generadores de casos de prueba constan, en general, de un resolutor de restricciones que genera datos concretos a partir de las restricciones de entrada.

El enfoque mas común adoptado a la hora de generar casos de prueba de forma estática consiste en realizar una *ejecución simbólica* del programa objetivo [10, 4] donde los contenidos de las variables son expresiones en vez de valores concretos. De esta forma, cada vez que la ejecución llega a un bifurcación, las condiciones para seguir por cada uno de los caminos se almacenan en forma de restricciones. Este proceso de ejecución simbólica se realiza hasta que se satisface el recubrimiento buscado. Una vez completado, se obtiene una serie de restricciones sobre la entrada a partir de las cuales se definen los casos de prueba.

1.4. Estado del Arte

En el ámbito de la generación automática de casos de prueba han surgido en los últimos años una serie de herramientas. A continuación se describen las más relevantes para contextualizar el presente proyecto.

1.4.1. PET

PET (Partial Evaluation-based Test Case Generator for Bytecode) [7, 1] es una herramienta cuyo propósito es generar casos de prueba de forma automática para programas escritos en *bytecode* (código de bytes de Java). PET adopta el enfoque previamente comentado, esto es, ejecuta el bytecode simbólicamente y devuelve como salida un conjunto de casos de prueba (test-cases). Cada caso de prueba está asociado a una rama del árbol de ejecución construido de acuerdo con el criterio de recubrimiento y se expresa como un conjunto de restricciones sobre los valores de entrada y una descripción de los contenidos de la memoria dinámica (o *heap*). Las restricciones de la memoria dinámica imponen condiciones sobre la

forma y contenidos de las estructuras de datos del programa alojadas en esta misma. PET utiliza de un resolutor de restricciones que genera valores concretos a partir de estas restricciones, permitiendo la construcción de los tests propiamente dichos.

PET puede usarse a través de una interfaz de línea de comandos o bien usando una interfaz web. Además soporta una variedad de opciones interesantes, como la elección de criterios de recubrimiento o la generación de tests *jUnit*. Estas opciones se describen con más detalle en el segundo capítulo.

1.4.2. PEX

PEX [14] es una herramienta desarrollada por Microsoft para la generación automática de casos de prueba. Este sistema ha sido desarrollado para la plataforma de desarrollo .NET y puede analizar cualquier programa que se ejecute en una máquina virtual de .NET.

PEX adopta un enfoque algo distinto al de PET al combinar la ejecución simbólica con la ejecución concreta, que es la ejecución ordinaria del programa. Mediante un proceso iterativo, PEX realiza una ejecución concreta del método a analizar y examina la traza de ejecución buscando ramas no exploradas. Una vez ha encontrado una rama no explorada, usa la ejecución simbólica y un sistema resolutor de restricciones para generar valores concretos que exploren dicha rama. Este proceso se repite hasta obtener el recubrimiento deseado. PEX puede guiarse por criterios de recubrimiento de instrucciones y de condiciones. El sistema puede además comprobar aserciones en el código que hacen el papel de oráculo para detectar errores directamente.

El hecho de combinar la ejecución simbólica con la concreta permite en muchos casos incrementar la escalabilidad y tratar con situaciones donde la ejecución no depende sólo del código sino de factores externos. Se entiende por factores externos el uso de librerías nativas, llamadas al sistema operativo o interacciones con el usuario. Ante este tipo de situaciones, la ejecución simbólica presenta grandes limitaciones y en general, no se puede aplicar.

Este sistema se puede integrar como un añadido (Add-on) en el entorno de desarrollo *Visual Studio* facilitando su uso enormemente.

1.4.3. CUTE

CUTE (Concolic Unit Testing Engine) [13, 12] es otro sistema de generación de casos de prueba basado en la ejecución concólica, que es la combinación de ejecución concreta y simbólica. Este sistema combina de una forma similar a PEX los dos tipos de ejecuciones y encuentra su mayor limitación en el resolutor de restricciones. En los casos en lo que el resolutor de restricciones no tiene potencia suficiente, parte de las variables simbólicas son sustituidas por valores concretos simplificando así el sistema a resolver.

CUTE ha sido desarrollado para C y se ha extendido recientemente para Java (jCUTE). Mientras CUTE analiza programas secuenciales en C, jCUTE analiza programas concurrentes en Java.

jCUTE consta de una interfaz gráfica independiente, permite la generación de tests jUnit y la visualización gráfica de las planificaciones concurrentes exploradas. También cabe mencionar la capacidad de la herramienta para comprobar aserciones y la posibilidad de combinarla con otras para la detección de errores de forma directa.

1.4.4. KeY

KeY [6] es un demostrador automático de teoremas para programas Java, basado en la lógica dinámica JavaCard. Para la demostración de teoremas, KeY utiliza un cálculo cuyas reglas realizan ejecución simbólica del programa. De manera colateral, la ejecución simbólica se puede utilizar para generar casos de prueba. En este sentido, los casos de prueba que genera Key están basados en verificación (*verification-based test generation*). Combina ejecución simbólica con verificación basada en modelos con precondiciones y postcondiciones.

Esta herramienta es capaz de generar *unit tests* para programas Java y permite visualizar los caminos generados por la ejecución simbólica.

1.5. Objetivos

El objetivo de este proyecto es la creación de una herramienta de generación de casos de prueba automática para programas Java. Se pretende

conseguir un sistema que esté a la altura de los ya existentes e incluso los mejore en algunos aspectos. La herramienta no sólo debe generar casos de prueba sino que debe permitir una fácil interpretación de estos para proveer al programador de una ayuda real durante el desarrollo del software. De hecho, el presente proyecto hace especial hincapié en el tratamiento posterior de la información generada con el objetivo de presentarla al usuario de una forma fácil de entender. Si el usuario es capaz de comprender los resultados de la herramienta rápidamente, podrá sacar más partido de estos. Las representaciones gráficas y la interacción con el usuario son los pilares fundamentales en los que se ha basado el desarrollo del proyecto.

Este proyecto es muy ambicioso y posee una gran envergadura que supera las posibilidades de un proyecto de fin de carrera en caso de ser abordado desde cero. En cambio, sí es posible realizar contribuciones interesantes si se hace uso de los medios y las herramientas existentes.

Este proyecto saca provecho de varias librerías y herramientas que facilitan la implementación. Entre estas se encuentran:

- *JGraph*: librería para el dibujado y la manipulación de grafos.
- *BCEL*: librería que procesa y extrae información de archivos de bytecode (.class).
- *ANTLR*: una herramienta para la generación automática de procesadores del lenguaje a partir de una gramática.

El uso de estas herramientas se explica en profundidad en los capítulos tres, cuatro y cinco respectivamente. Además, se parte de una herramienta ya existente que funciona sobre bytecode. Esto nos evita repetir trabajo ya hecho y nos permite llegar mucho más lejos.

Como núcleo de la aplicación y punto de partida se ha elegido PET. Esta elección se ha basado fundamentalmente en dos razones: PET es software libre, lo que permite introducir modificaciones fácilmente y, en segundo lugar, dispone de una interfaz de línea de comandos sencilla e independiente, permitiendo su adaptación e integración dentro de la aplicación.

El proyecto se ha denominado jPET, ya que a diferencia de PET, que trabaja sobre bytecode, está orientado exclusivamente para el análisis de programas Java.

1.6. Contribuciones

En este proyecto se presenta jPET, una herramienta de generación automática de casos de prueba integrada en el entorno de desarrollo Eclipse con el propósito de ayudar a los desarrolladores a probar sus programas Java durante el proceso de desarrollo del software. Esta integración se ha llevado a cabo implementando jPET como un plugin de Eclipse. Las principales funcionalidades de jPET, que son las contribuciones más importantes de este proyecto, se resumen a continuación:

- Permite ejecutar PET desde la interfaz de Eclipse y seleccionar los argumentos y los métodos a ejecutar a través de una ventana de diálogo. Esta funcionalidad se explica en el segundo capítulo.

- Incorpora un visor de casos de prueba (test-case viewer) para visualizar la información relativa a los casos de prueba. El visor muestra los contenidos de la memoria global representando las estructuras de datos alojadas en la memoria dinámica y sus relaciones (denominado *heap* a partir de este punto). Puede mostrar el contenido de la memoria antes de la ejecución (heap de entrada) y después de la ejecución de cada caso de prueba (heap de salida). En el tercer capítulo se detalla el funcionamiento e implementación de este visor.

- Puede mostrar la traza de ejecución de un caso de prueba dado (i.e., la secuencia de instrucciones que el caso de prueba ejecutaría) de dos formas distintas: (1) Bien marcando todas las instrucciones implicadas o bien (2) permitiendo al usuario reproducir la secuencia de instrucciones paso a paso usando la interfaz de depuración de Eclipse. Los detalles acerca de dicha funcionalidad son tratados en el capítulo cuarto.

- Por último, puede analizar sintácticamente precondiciones de métodos escritas en JML (Java modeling language) y usarlas para evitar la generación de casos de prueba poco interesantes. El capítulo quinto se centra en esta funcionalidad.

Estas funcionalidades cumplen con los objetivos buscados. La integración de jPET en Eclipse consigue un sistema útil y práctico desde el punto

de vista del programador. El visor de casos de prueba y la funcionalidad para mostrar la traza de ejecución facilitan la comprensión de los casos de prueba y su relación con el código fuente respectivamente. Finalmente, el sistema de análisis de precondiciones permite al programador filtrar la información que generará la herramienta, simplificando el análisis posterior y aumentando la productividad.

Capítulo 2

El plug-in de Eclipse para jPET

Este capítulo proporciona una visión global de jPET. Se detalla el funcionamiento general de la herramienta, su forma de uso y su instalación y termina mostrando un ejemplo de un programa Java que se usará a lo largo del documento para ilustrar las diferentes funcionalidades de jPET.

2.1. El Plug-in para Eclipse

Eclipse es un IDE muy potente para el desarrollo de aplicaciones en diferentes lenguajes. Está construido mediante un núcleo al cual se le acoplan diferentes plug-ins, es decir, módulos que implementan funcionalidades añadidas a la aplicación. Gracias a su arquitectura, es posible desarrollar plug-ins que sólo dependan del núcleo de Eclipse, o bien que dependan a su vez de otros plug-ins. Este modelo de capas se consigue gracias a los llamados *puntos de extensión* que permiten interconectar los plug-ins unos con otros para, entre otras cosas, aportar algo a la interfaz, procesar los resultados obtenidos por el plug-in anfitrión, etc. En la imagen 2.1 se puede visualizar esquemáticamente cómo un plug-in extiende a otro haciendo uso de sus puntos de extensión.

A la hora de desarrollar un plug-in para Eclipse es importante definir los puntos de extensión que queremos que éste tenga si queremos que pueda ser ampliado. Además es importante estudiar cuales son los puntos de ex-

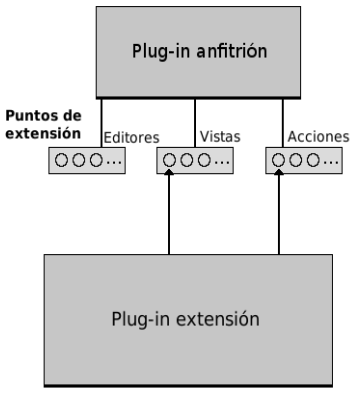

Figura 2.1: Esquema del uso de puntos de extensión para plug-ins

tensión de los módulos que queremos usar para contar con las posibilidades y limitaciones de cada uno.

2.2. Ejecución de jPET

Para elegir los métodos que se quiere analizar se han de seleccionar desde la vista *Outline*. Como puede verse en la Fig. 2.2, jPET añade un botón a la barra de herramientas de Eclipse y al menú de Eclipse. Al hacer click en él, una ventana de diálogo permite a los usuarios seleccionar las preferencias y ejecutar jPET. Esta opción permite establecer opciones internas tales como el criterio de recubrimiento, el rango de variables numéricas, el directorio de salida donde los archivos JUnit/XML son generados, etc. Estas preferencias se explican en detalle más adelante.

Una vez que se ha mandado a analizar algún método es necesario abrir la vista propia de jPET disponible en *Window - Show View - Other... - JPet - JPet view*. Es posible ver la vista en la zona inferior de la imagen 2.2. En ella se muestran los casos de prueba generados para el proyecto seleccionado en el *Package Explorer* ordenados por paquetes, clases y métodos. En los capítulos posteriores se explican en detalle las opciones disponibles desde esta vista.

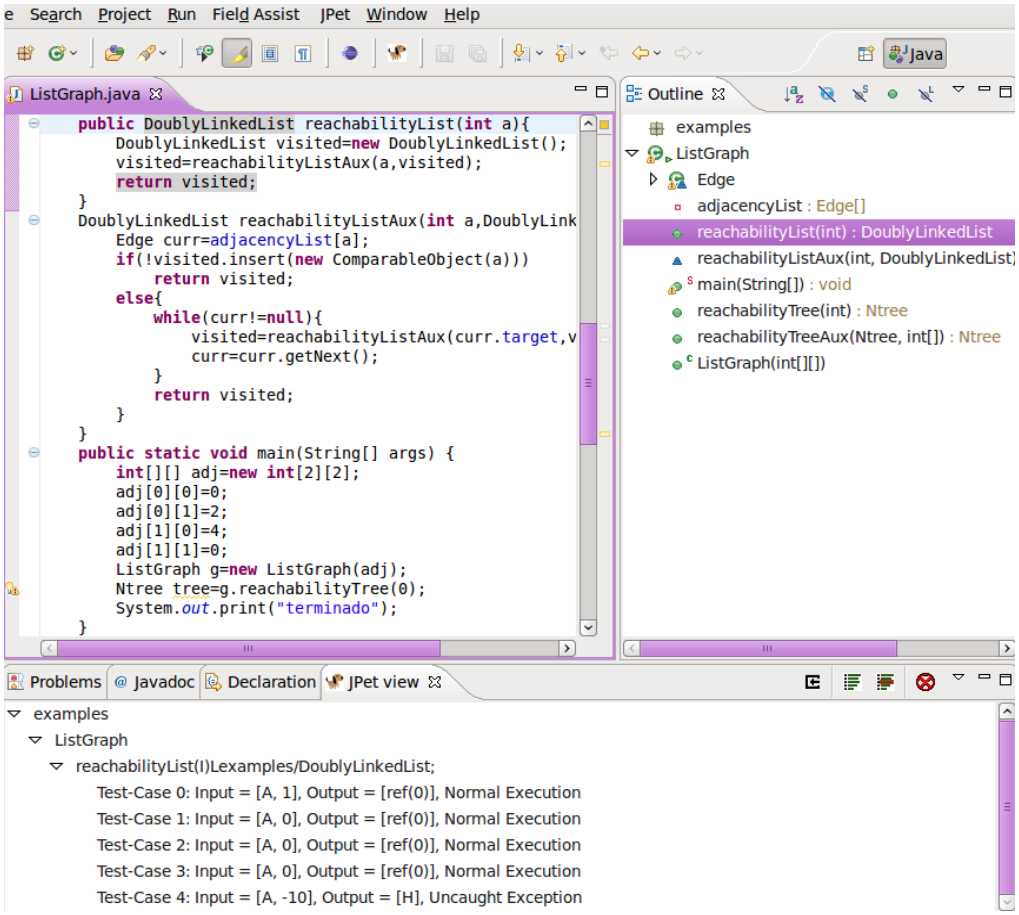

Figura 2.2: Vista jPET y selección de métodos en la vista Outline

2.2.1. Enlazando con PET

Como ya se ha comentado, jPET una parte del proyecto ha consistido en enlazar jPET con PET para la generación de los casos de prueba sobre código de bytes, para lo cual hay que recoger todas las opciones elegidas por el usuario y transformarlas al formato adecuado. En concreto, al hacer uso de PET en modo consola, hay que traducir las opciones del usuario a una línea de texto que recoge todas las preferencias.

Es posible que para realizar el proceso de testing de una aplicación sea necesario analizar una gran cantidad de clases y métodos y es importante que, durante este proceso, el programador pueda seguir trabajando sin in-

terrupción. Para hacer pleno uso de las funcionalidades que nos aporta Eclipse decidimos extender la clase Job, la cual nos permite mandar ejecutar diversas tareas y tener control sobre cada una desde la interfaz. Eclipse nos permite ver el estado de cada tarea, su evolución mediante una barra de progreso e incluso detener la que deseemos. En la figura 2.3 se puede ver un ejemplo de la ejecución de jPET.

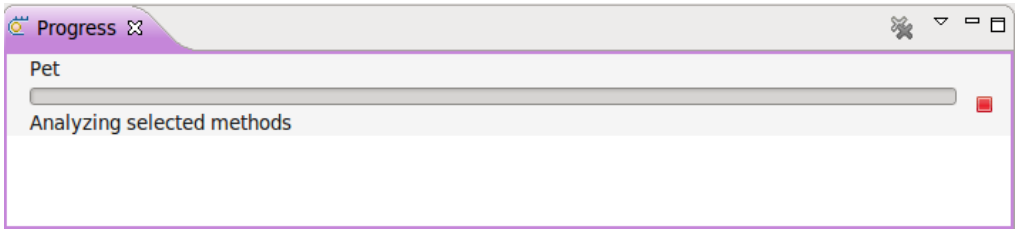

Figura 2.3: Ejecución de jPET

2.2.2. Fichero XML

Los resultados arrojados por PET se devuelven en modo consola con un formato propio y difícilmente tratable. Por ello, decidimos modificar la herramienta para que generase un fichero XML con toda la información que necesitamos.

Como se verá más adelante este fichero XML es parte fundamental de jPET y se usa tanto en el visor de casos de prueba como en los dos métodos de visualización de trazas. Fue necesario modificar el código de PET para generar el fichero XML con toda la información que necesitábamos.

jPET crea, dentro de la carpeta especificada por el usuario en las preferencias, las carpetas necesarias para almacenar y organizar los ficheros XML separados por clases y paquetes.

A grandes rasgos, cada fichero XML tiene la información de todos los casos de prueba generados para un método. En concreto, para cada caso de prueba el fichero contiene la siguiente información:

- **Firma**: La firma del método para el cual se ha creado el caso de prueba.

- **Argumentos de entrada**: Argumentos de entrada para el caso de prueba, puede contener punteros a objetos del heap o bien valores concretos.

- **Heap de entrada**: Muestra información de cada objeto del heap de entrada, su tipo, información de sus elementos si es un array y de sus campos si es un objeto, etc.

- **Heap de salida**: Ídem para el heap de salida.

- **Retorno**: Valor de retorno o referencia al objeto que se devuelve en el caso de prueba.

- **Flag de excepción**: Flag de excepción que se activa si el caso de prueba genera una excepción no capturada.

- **Traza**: Traza del caso de prueba, guarda la información del nombre de las instrucciones que se ejecutan y su contador de programa para identificar cada instrucción inequívocamente.

Una vez generado el fichero es necesario extraer toda esa información para poder usarla en jPET, para ello hemos usado el API que ofrece Java para este propósito incluida en el paquete *org.w3c.dom* que brinda clases para parsear y almacenar toda la información contenida en un fichero XML.

2.3. Preferencias

PET consta de multitud de parámetros configurables que permiten la generación de casos de prueba ajustados a la necesidad del usuario. Además jPET incorpora y amplía estas opciones para ofrecer una experiencia más completa al usuario. Las principales opciones de PET son:

- **Coverage criterion**: Permite seleccionar el criterio de recubrimiento. Existen dos criterios implementados, el Block-k y el Depth-k. Ambos requieren de un parámetro adicional que indica el número de veces que cada bloque ha de ser explorado en el primer criterio y la profundidad del árbol en el segundo.

- **Numeric test-cases or path-constraints**: Este parámetro permite elegir si se quieren obtener casos de prueba especificados como conjuntos de restricciones o como valores numéricos concretos. En el caso de que se quieran valores numéricos es necesario establecer un rango de valores posibles.

- **Labeling Strategy**: Este parámetro permite transformar los *path-constraints* en valores concretos que satisfacen las restricciones utilizando la función *labeling* existente en *CLP-FD*, *Constraint Logic Programming over Finite Domains*, una librería de *Prolog* que permite describir condiciones que una solución debe satisfacer.

- **References aliasing**: Indica si se permite la posibilidad de que existan alias, es decir, referencias distintas que apunten a un mismo objeto. En general, al activar esta opción se generan multitud de casos de prueba adicionales, que a veces puede ser interesante.

- **Generate JUnit test**: Si está activada esta opción, una serie de tests usando JUnit son generados en base a los casos de prueba que PET genera.

- **Tracing**: Esta opción permite obtener la traza de ejecución de cada uno de los casos de prueba generados.

Además de estas opciones, jPET añade las siguientes preferencias:

- **JUnit Path**: Permite especificar el directorio en el que se generarán los ficheros JUnit generados.

- **Show JUnit Test**: Cuando esta opción está marcada, al finalizar el análisis, se abrirá automáticamente en el editor el fichero con las pruebas de JUnit.

- **XML files Path**: Ofrece la posibilidad de elegir dónde se crearán los ficheros XML generados por la herramienta.

Cabe mencionar que pese a que no sean visibles de forma directa, las siguientes preferencias también se han añadido en jPET para integrar alguna extensión o bien facilitar la labor del usuario:

- **Classpath**: Permite especificar un directorio, diferente al que hay por defecto, donde encontrar la clase a analizar. Desde el punto de vista del usuario esta opción le posibilita poder trabajar en cualquier directorio.

- **Precond**: Envía las precondiciones a PET para que las tenga en cuenta a la hora de realizar el análisis. En el capítulo 5 se habla en detalle sobre el uso de precondiciones en jPET.

2.4. Instalación de jPET

En un intento de integrar jPET completamente en Eclipse hemos optado por usar el sistema de instalación de software del famoso IDE. Para instalar jPET tan sólo es necesario abrir el asistente de software nuevo de Eclipse pulsando en el menú *Help* y después en *Install New Software...* (los nombres de los botones puede variar dependiendo de la versión y el idioma de Eclipse). En el campo *Work with* hay que escribir la URL de la descarga del software, para nuestro caso es:

`http://costa.ls.fi.upm.es/pet/petUpdateSite/`

Opcionalmente se puede añadir la dirección de descarga para futuras instalaciones de jPET, para ello hay que pulsar en el botón *Add...* que hay a la derecha y pulsar en el botón *OK* del cuadro emergente. Opcionalmente se puede asignar un nombre a la dirección para reconocerla en posteriores usos. En la imagen 2.5 se muestra el proceso explicado gráficamente. Una vez que jPET aparece en la lista de software, basta con marcarlo y seguir el proceso de instalación como cualquier otra aplicación para Eclipse. El último paso de la instalación es reiniciar Eclipse para que se apliquen los cambios.

Cabe destacar que al instalar jPET ya se incluye un ejecutable de PET y los ficheros necesarios para el resto de librerías por lo que no es necesario realizar más pasos que los aquí indicados para disfrutar plenamente de la aplicación.

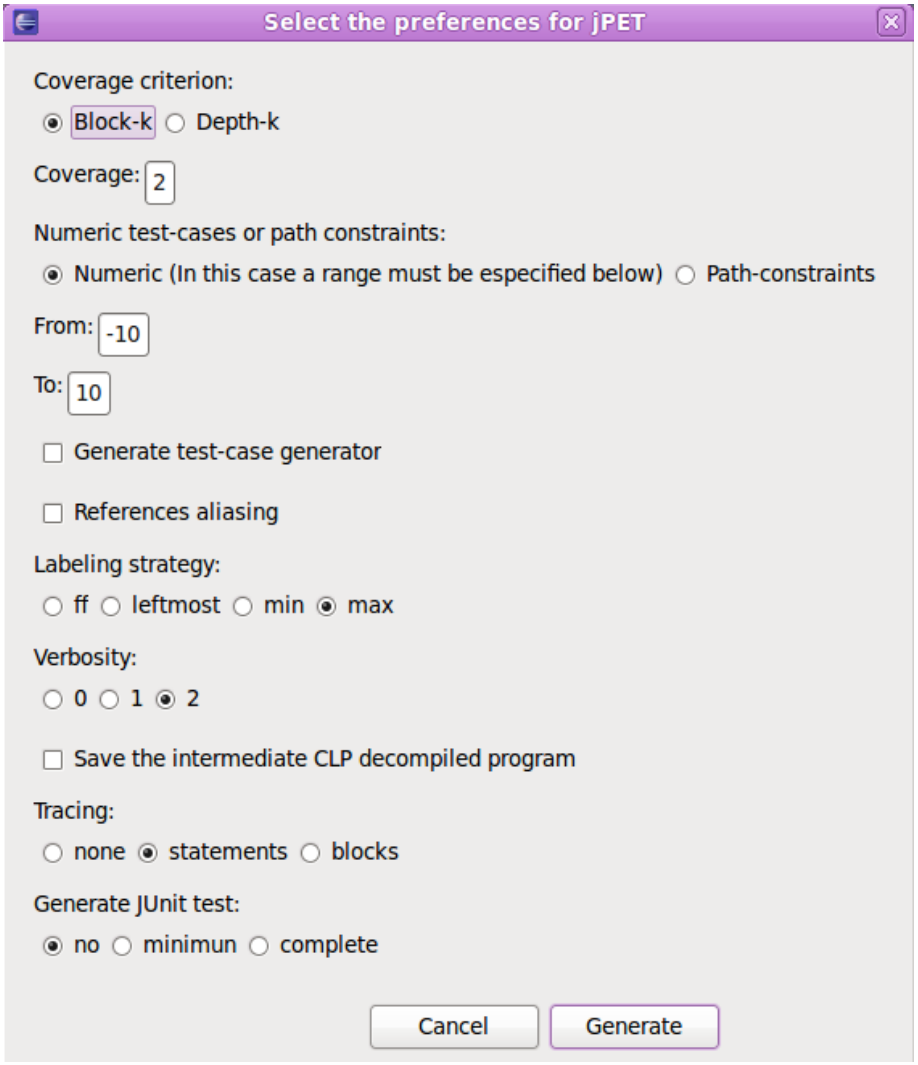

Figura 2.4: Diálogo de preferencias de jPET

2.5. Ejemplo ilustrativo

Vamos a ilustrar la funcionalidad de jPET a través de un algoritmo sencillo, `reachabilityList` de la clase `ListGraph`. Esta clase implementa grafos dirigidos usando listas de adyacencia, una representación muy útil para grafos dispersos. El único atributo de `ListGraph` es un vector de listas llamado `adjacencyList`. Cada elemento, `adjacencyList[i]`, contiene

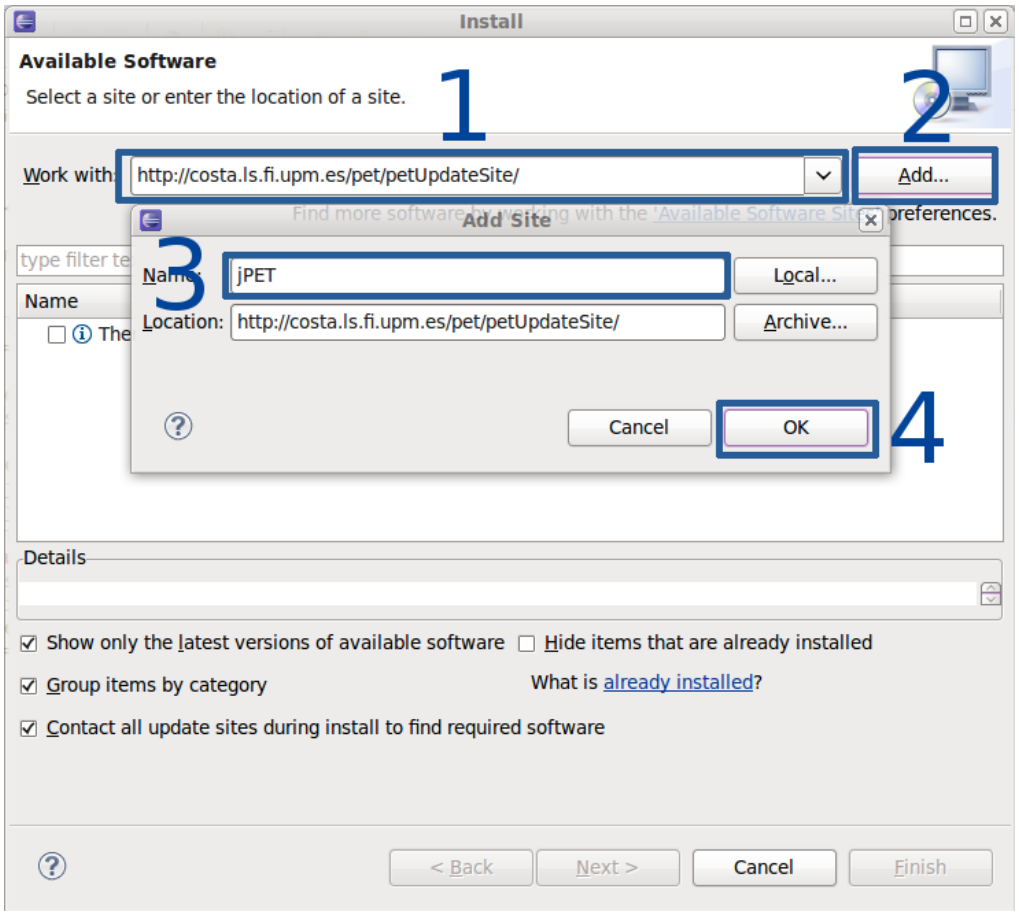

Figura 2.5: Instalación de jPET

una lista de aristas que parten del nodo `i`. Cada arista contiene dos enteros que representan el nodo destino y el coste de la arista. Además, cada arista tiene a su vez una referencia a la siguiente arista de la lista. Esta referencia será null si nos encontramos ante la última arista. El método que va a ser probado, `reachabilityList` de la Fig. 2.6, crea una lista ordenada doblemente enlazada (`DoublyLinkedList`) compuesta por los nodos alcanzables desde un nodo dado cuyo identificador es un parámetro de entrada. Cabe mencionar un método auxiliar, `insert` en la Fig. 2.7 de la clase `DoublyLinkedList`. Un objeto `DoublyLinkedList` contiene una referencia al primer elemento (`first`), al último (`last`) y al elemento actual

(`current`). El elemento actual es un puntero que permite acelerar el acceso a los elementos de la lista.

El método `insert` usa el método `search` que recibe un elemento y si éste se encuentra en la lista, mueve el puntero `current` para que apunte a dicho elemento. Si, por el contrario, el elemento no está en la lista, entonces `current` apuntará al nodo previo al lugar que el elemento debía ocupar, con la excepción de que el elemento debiera ser el primero. En ese caso, `current` apuntará al primer nodo de la lista.

El algoritmo principal (`reachabilityList`) recibe un identificador de un nodo y explora sus aristas. Para cada arista, añade su nodo destino a la lista de salida y explora recursivamente el propio nodo destino siempre y cuando no haya sido explorado ya. El algoritmo de inserción en la lista (`insert`) es capaz de detectar duplicados. Esto facilita la detección de nodos que ya se han inspeccionado.

```
public DoublyLinkedList reachabilityList(int a){
   DoublyLinkedList visited=new DoublyLinkedList();
   visited=reachabilityListAux(a,visited);
   return visited;
}
DoublyLinkedList reachabilityListAux(int a,
                     DoublyLinkedList visited){
   Edge curr=adjacencyList[a];
   if(!visited.insert(new ComparableObject(a)))
      return visited;
   else{
      while(curr!=null){
         visited=reachabilityListAux(curr.target,visited);
         curr=curr.getNext();
      }
      return visited;
   }
}
```

Figura 2.6: Algoritmo de ejemplo: ListGraph.reachabilityList

En nuestro ejemplo, PET es ejecutado con un criterio de recubrimiento loop-2, que limita a un máximo de 2 iteraciones por bucle, y un rango de

```
public boolean insert(ComparableObject content){
   Node node=new Node(content);
   if (current==null){
      current=node; first=node; last=node;
      node.setNext(null); node.setPrevious(null);
      return true;
   }else{
      search(content);
      if(current.content.compareTo(content)<0){
         node.setNext(current.getNext());
         node.setPrevious(current);
         if(current.getNext()!=null)
            current.getNext().setPrevious(node);
         current.setNext(node);
         return true;
      }else{
         if(current.content.compareTo(content)==0){
            current.content=content;
            return false;
         }else{
            node.setPrevious(current.getPrevious());
            node.setNext(current); current.setPrevious(node);
            current=node; first=node;
            return true;
}}}}
```

Figura 2.7: Método con un bug: DoublyLinkedList.Insert

variables numéricas de -10 a 10. Se generan cinco casos de prueba para el método `reachabilityList` que están listados en la vista de jPET (jPET view) en la parte inferior de la Fig. 2.2. Estos casos de prueba pueden ser examinados como se indica más adelante.

Capítulo 3

Visor de casos de prueba

Como se ha mencionado en la introducción, dado un método, PET genera descripciones de los casos de prueba. Cada caso de prueba se especifica en forma de restricciones de los valores de entrada y una descripción de los objetos alojados en memoria antes y después de la ejecución. Las descripciones generadas pueden ser incompletas, pues sólo especifican los requisitos estrictamente necesarios para la consecución del caso de prueba concreto, pudiendo quedar partes de las estructuras de datos indeterminadas.

```
Input: Args = [ref(A), 1] Heap = heap([ (A,object(
'examples/ListGraph',[field('examples/ListGraph.adjacencyList
[Lexamples/ListGraph\$Edge;',ref(B))|C])),(B,array(D,2,
[null,ref(E)|F])),(E,object('examples/ListGraph\$Edge'
[field('examples/ListGraph\$Edge.target:I',0),field(
'examples/ListGraph\$Edge.next:Lexamples/ListGraph\$Edge;'
,null)\|G]))\|H],I)
```

Figura 3.1: Salida de Pet: heap de entrada del primer caso de prueba generado del método `reachabilityList`

La salida de PET incluye una lista de casos de prueba especificados en forma de texto con el formato visible en el ejemplo 3.1, una descripción del heap de entrada para el método `reachabilityList`. Esta representación se vuelve compleja y difícil de entender incluso para ejemplos sencillos.

El visor de casos de prueba (test-case viewer) de jPET resuelve este problema permitiendo al programador la posibilidad de navegar por las representaciones de los heaps de forma gráfica. Para acceder a esta funcionalidad basta con hacer click derecho sobre el caso de prueba deseado en la vista de jPET y seleccionar la opción `Show Test Case` como puede verse en la Fig. 2.2.

3.1. Descripción y funcionamiento

La imagen 3.2 muestra la interfaz general del visor para nuestro código de ejemplo. Como puede apreciarse, éste consta de tres áreas numeradas que se describen a continuación:

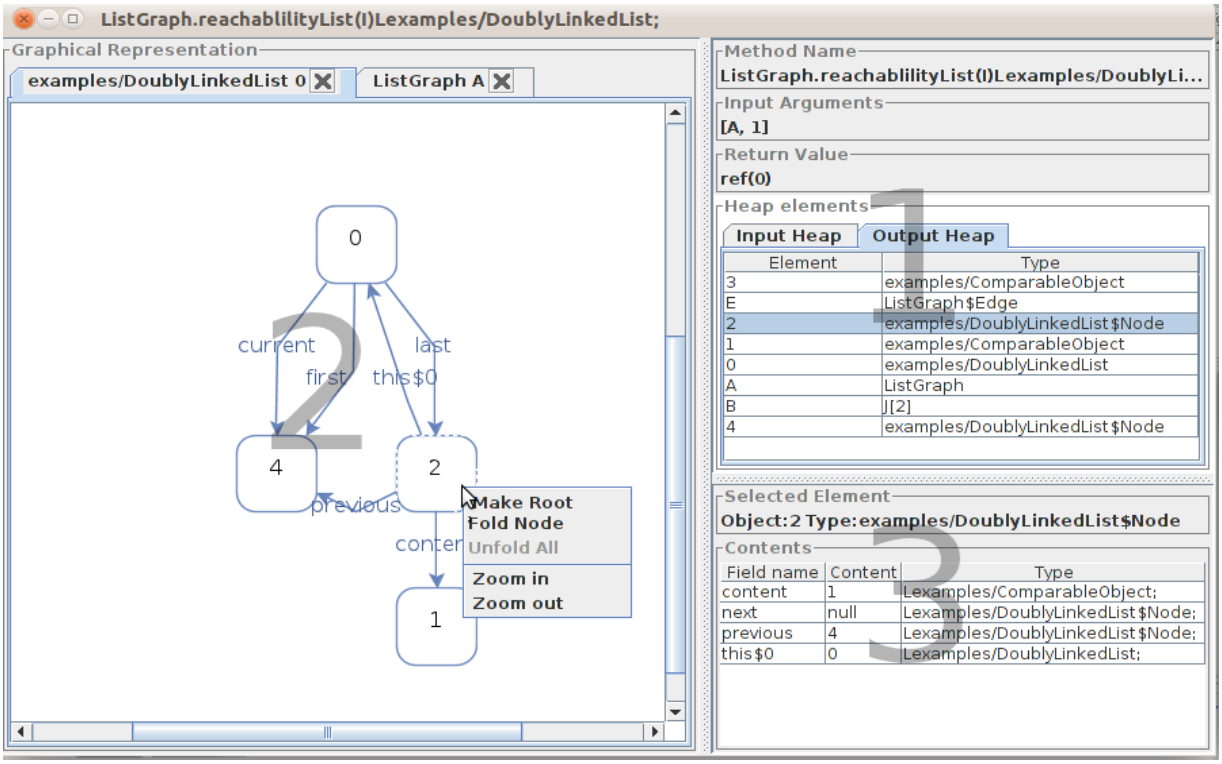

Figura 3.2: Interfaz del visor de casos de prueba

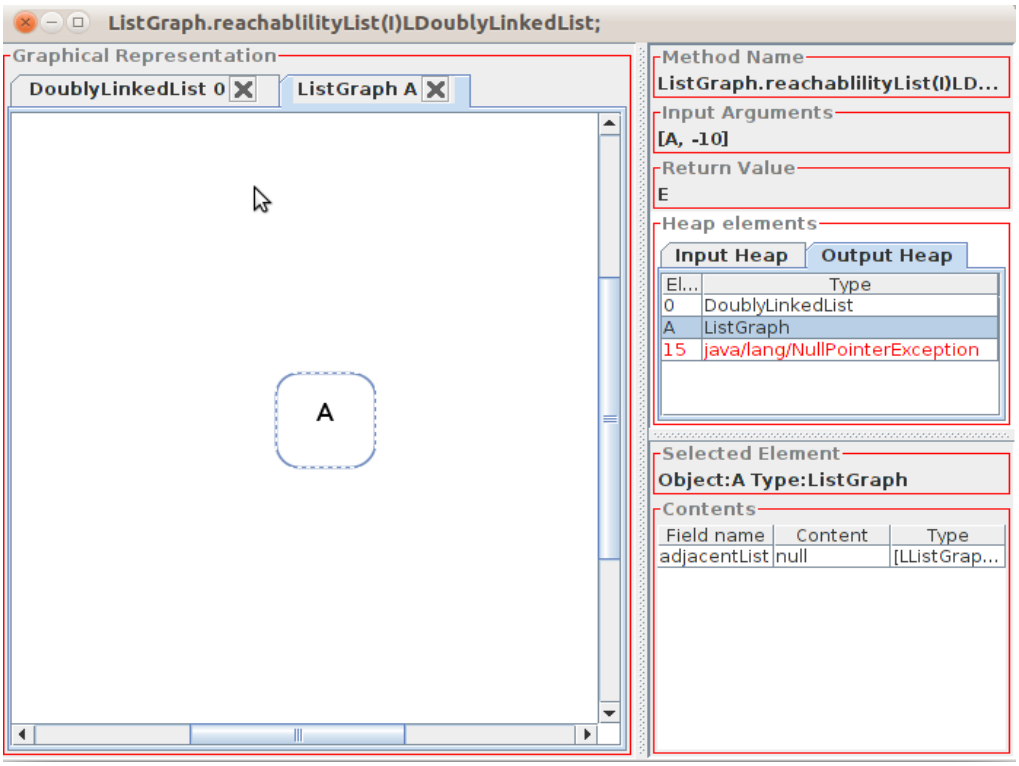

Figura 3.3: Interfaz del visor de casos de prueba en un caso acabado en excepción

3.1.1. Área 1: Descripción del caso de prueba

El área superior derecha contiene el nombre del método, los argumentos de entrada, el valor de salida, y la lista de objetos presentes en el *heap* de entrada y salida (separados en dos pestañas denominadas Ïnput Heapz .ºutput Heaprespectivamente). Además, si el método analizado ha terminado con una excepción, los bordes de las distintas áreas de la interfaz aparecerán en color rojo, así como el objeto que representa la excepción, como puede verse en la Fig. 3.3.

Los objetos se muestran en una tabla donde cada fila representa un objeto que consta de dos campos. El primer campo es el nombre del objeto, un identificador único asignado por PET y el segundo campo es el tipo del

objeto, es decir, la clase a la cual pertenece.

Éste es el punto de partida desde donde el programador puede decidir qué objetos explorar. Haciendo click se puede ver el contenido del objeto en el área 3 y haciendo doble click (o click derecho y seleccionando la opción `Load Graph`) una nueva pestaña se abrirá en el área 2, el visor del heap, con el objeto seleccionado como raíz de la representación.

3.1.2. Área 2: Reprensentación gráfica

Este área contiene la esencia de la herramienta, ya que es la que representa los objetos y sus relaciones de una forma fácil e intuitiva.

Un *heap* se puede representar como un grafo. Pero un grafo representado de forma arbitraria no suele corresponderse con la concepción del programador. Mostrar todos los objetos y sus referencias mutuas puede dar lugar a grafos extremadamente complicados, por lo que es necesario establecer cierto orden en la reprentación.

Habitualmente, cuando el programador concibe las relaciones entre los objetos suele hacerlo con una jerarquía implícita. Hay objetos contenidos dentro de otros y no todos tienen la misma importancia. En la práctica, esta jerarquía viene dada sólo por el significado que el programador da a cada uno de los objetos y no aparece de forma explícita en la representación. Por ejemplo, es muy común que haya objetos hijos que tengan como referencia a sus padres, con lo que el grafo de relaciones entre objetos no es acíclico. En consecuencia, es muy difícil inferir orden alguno.

La solución adoptada está inspirada en la representación habitual de unas herramientas familiares a los programadores, los depuradores. Estos permiten al programador elegir un objeto a explorar, expandirlo y ver así los objetos referenciados por éste. A continuación, el programador puede expandir cualquiera de los objetos hijos y ver su contenido. De esta forma el heap se representa como un árbol posiblemente infinito que puede ser explorado hasta donde se quiera y donde la raíz ha sido elegida conscientemente por el programador.

En la presente herramienta, el usuario comienza seleccionando un objeto en el área 1 (descripción del caso de prueba). A partir de dicho objeto se crea una representación gráfica donde se muestra ese único objeto como raíz. Por defecto, al iniciar el visor se crea una representación gráfica por

cada objeto que sea argumento de entrada. El usuario puede expandir el nodo, mostrándose así los objetos referenciados por éste. Cada uno de estos nuevos nodos puede ser expandido a su vez. Los nodos también pueden ser contraídos para ocultar sus hijos. La representación tiene estructura de árbol salvo una excepcioń, si un objeto ya se encuentra representado por un nodo y le corresponde ser expandido, no se crea un nodo duplicado, sino que se crea la arista ascendente correspondiente. Con esto se consigue una representación finita, jerárquica en la que cada objeto se corresponde con un único nodo. Por defecto, las representaciones gráficas se muestran inicialmente totalmente expandidas.

Este mecanismo implica que algunos nodos pueden, por ejemplo, cambiar de padre y la representación final no es independiente del orden de expansión de los nodos. Cada representación nos muestra el heap desde una perspectiva distinta.

La Fig. 3.1.2 es un ejemplo del comportamiento de las representaciones. La representación es una lista doblemente enlazada donde el nodo 0 es la lista en si; los nodos 2, 4, y 950 son los elementos de la lista en ese orden y los objetos 1 y 1123 son los contenidos de los elementos 2 y 950 respectivamente.

En la expansioń del nodo 2 se generan dos hijos y una arista ascendente. En la expansión del nodo 950 en cambio, se genera un único hijo nuevo, pero se añade una referencia al nodo 4 previamente creado (además de otra arista ascendente). Finalmente, al contraer el nodo 2, el nodo 1 desaparece al quedarse sin padre y el nodo 4, al recibir otra referencia, cambia de padre, pasando a ser hijo del nodo 950.

Los nodos contienen el nombre identificador del objeto y las aristas tienen una etiqueta con el nombre del atributo que representan. En el caso de los vectores o matrices las aristas representan el índice de la posición en el mismo.

La herramienta permite a su vez:

- Expandir todos (unfold all), expande el nodo dado y todos sus descendientes de forma recursiva. Se puede realizar esta acción haciendo doble click sobre el objeto.

- Convertir en raíz (make root), crea una nueva pestaña con una nueva representación cuya raíz es el objeto seleccionado.

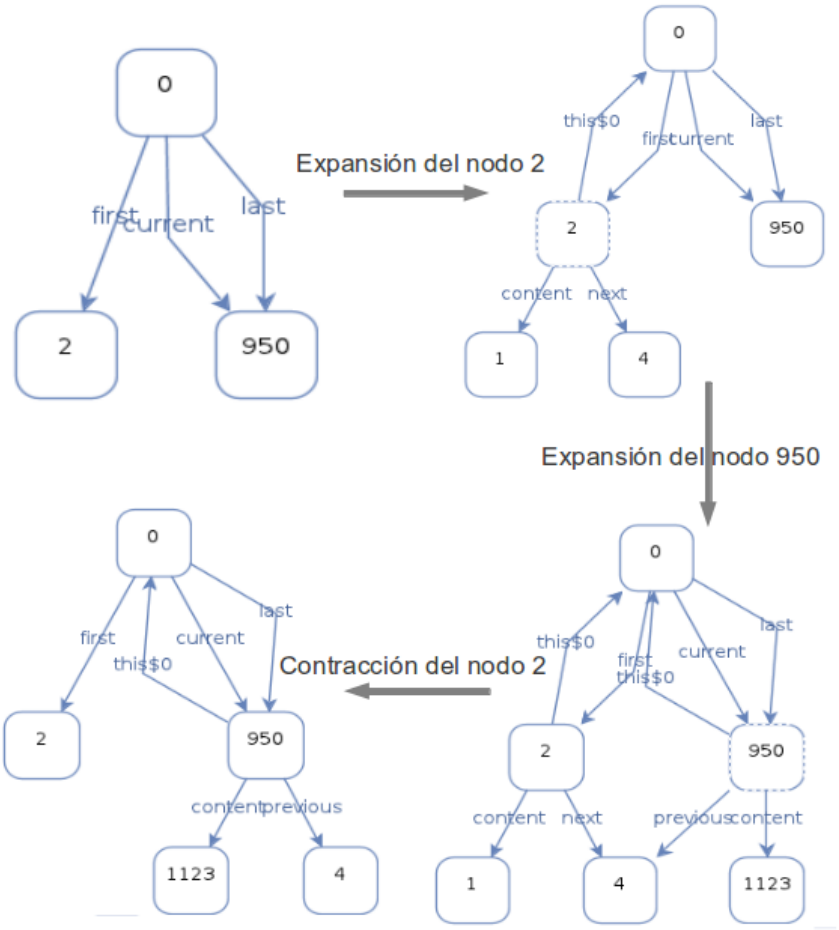

Figura 3.4: Secuencia de expansiones y contracciones de una representación de lista doblemente enlazada.

- Seleccionar un objeto (haciendo click sobre él), muestra su contenido en el área 3 (objeto seleccionado).

- Mover nodos y aristas, permitiendo al usuario adaptar la representación para un mayor entendimiento de ésta.

- Escalar la representación (Zoom in/Zoom out).

3.1.3. Área 3: Objeto seleccionado

La parte inferior derecha del visor contiene información sobre el objeto actualmente seleccionado. Muestra el nombre y tipo del objeto, así como los valores de sus atributos. Cada atributo consta de tres campos: el nombre del atributo, su contenido y su tipo de datos.

Esto permite al programador examinar cualquier objeto en detalle. Especialmente los valores de los atributos de tipo primitivo, tales como enteros, que no aparecen en la representación gráfica.

3.2. Uso del visor para encontrar errores

En las imágenes 3.5 y 3.6 pueden verse respectivamente las entradas y heaps de los casos de prueba 0 y 2 obtenidos para nuestro ejemplo principal.

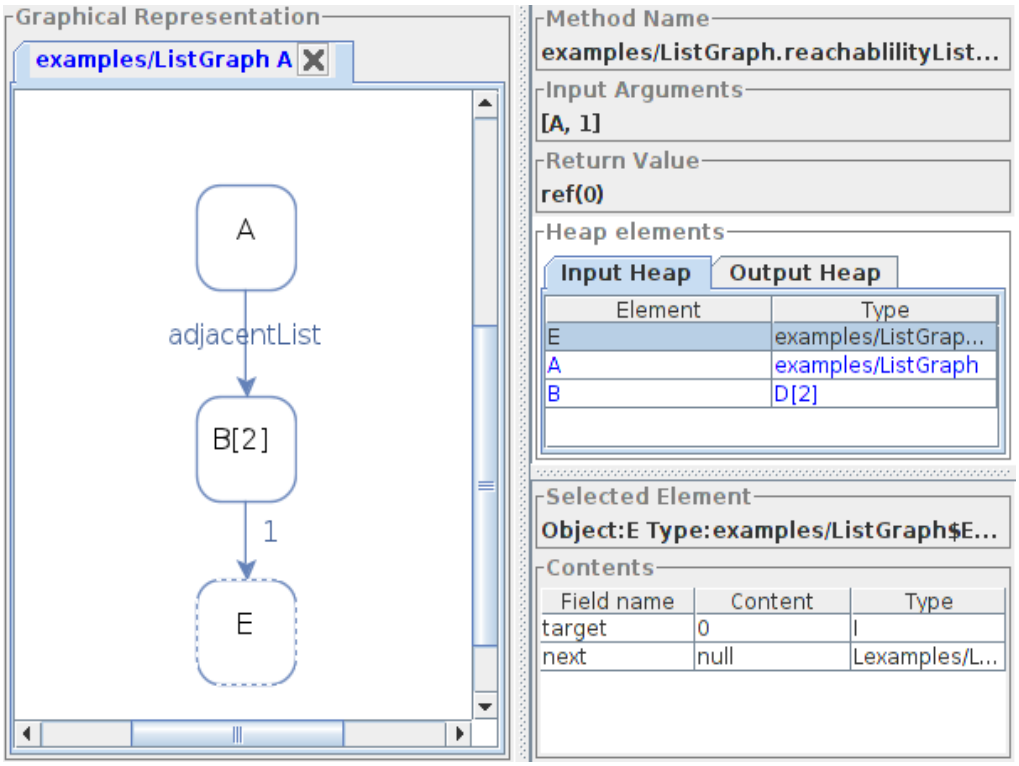

Figura 3.5: Ejemplo principal: heap de entrada del caso de prueba 0

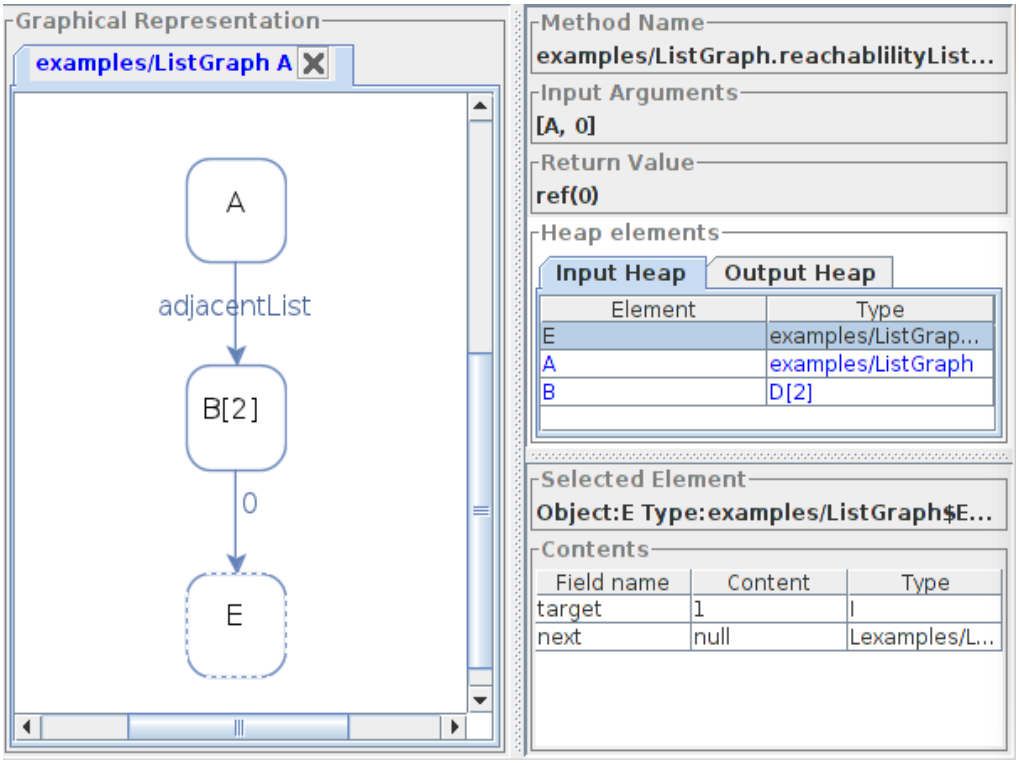

Figura 3.6: Ejemplo principal: heap de entrada del caso de prueba 2

En nuestro ejemplo, en el caso de prueba 0 (Fig. 3.5) podemos ver que la lista de los elementos adyacentes a 1 contiene una arista cuyo destino es 0. Como el argumento de entrada es 1, la lista de salida esperada es $[0, 1]$. En cambio, en el caso de prueba 2 (Fig. 3.6), la lista de adyacencia contiene un elemento en su índice 0 cuyo destino es 1. Es decir, la lista de salida esperada debería contener 0 y 1 también. Al ser las listas de salida ordenadas, ambas deberían ser iguales.

Fijémonos ahora en las representaciones de los heaps de salida de los casos de prueba 0 y 2 en la Fig. 3.7. Mostramos ambos grafos desde la perspectiva de las listas de salida (etiquetadas con 0) y expandimos las representaciones. A primera vista, es evidente que las dos estructuras son diferentes. Después de un análisis más detallado de la estructura de 2, se puede ver que **el atributo `last` no está apuntando al último elemento de la lista**. Acabamos de encontrar un error en nuestro ejemplo. En la

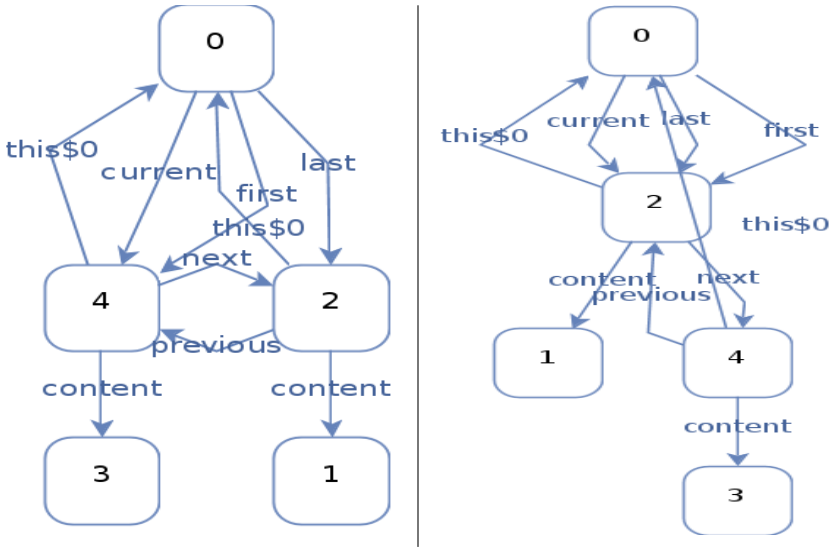

Figura 3.7: Ejemplo principal: heaps de salida de los casos de prueba 0 y 2

sección de visualización de la traza veremos como la correspondiente funcionalidad (el visor de trazas) puede ayudar al usuario a localizar el error en el código fuente.

3.3. Detalles de implementación

El visor de casos de prueba puede considerarse casi como una aplicación por derecho propio, ya que recibe como entrada las estructuras de datos que representan un caso de prueba y a partir de ahí tiene un funcionamiento totalmente autónomo. El visor sería, en consecuencia, muy fácilmente adaptable para funcionar integrado en otros entornos de desarrollo o incluso como una aplicación independiente.

3.3.1. Herramientas usadas

La implementación del visor de casos de prueba requiere una interfaz gráfica avanzada. También necesita una arquitectura que permita la visualización de grafos a partir de una lógica y el tratamiento avanzado de éstos.

Para poder abordar estos requerimientos fue necesario hacer un análisis de las ventajas e inconvenientes de las tecnologías existentes. Entre las librerías consideradas se encuentran `yWorks` y `Jung` pero finalmente se optó por `JGraph` [3]. Entre los criterios considerados se encuentra la potencia, flexibilidad, facilidad de uso y el hecho de que sean librerías de software libre o no.

`JGraph` es una librería de software libre (BSD) para la visualización y trazado de grafos implementada sobre `Swing` (una de las más famosas librerías gráficas sobre Java). Al estar basada en `Swing` hereda en gran parte su filosofía y características, su enorme flexibilidad y su potencia. `Swing` está escrita íntegramente en Java por lo que cualquiera de sus componentes puede ser personalizado. Además, ambas librerías gozan de una portabilidad excelente por esta misma razón.

A pesar de todas estas cualidades, se han tenido que afrontar importantes dificultades. En el momento de la implementación se acababa de publicar una nueva versión de la librería con una interfaz totalmente nueva. Después de sopesar los pros y los contras se optó usar la nueva versión para facilitar el futuro mantenimiento del sistema. Esta decisión ha supuesto problemas adicionales ya que la documentación estaba incompleta y la comunidad de usuarios inmadura, dificultando en gran medida su aprendizaje y su uso avanzado.

Finalmente, cabe mencionar un aspecto que, si bien no ha supuesto grandes dificultades, si ha condicionado en cierta manera la implementación de la herramienta. Eclipse está implementado usando SWT, una librería gráfica distinta a Swing. Aunque es posible integrar ambos sistemas, la integración es a menudo fuente de problemas [8] por lo que se ha optado por mantener el visor de casos de prueba en una ventana independiente. Esta independencia conlleva además la ventaja de poder abrir y explorar varios casos de prueba sin sobrecargar la interfaz propia de Eclipse.

3.3.2. Estructura general

El visor de casos de prueba consta de dos módulos principales:

- El módulo principal contiene una interfaz gráfica basada en `Swing` que permite a los programadores interactuar con el `Graph Manager` (el módulo encargado del manejo de grafos) creando, mostrando y

explorando las representaciones del heap que ellos elijan. Esta capa implementa las distintas áreas de la interfaz y sirve como nexo entre ellas.

Este módulo recibe como parámetro un objeto que representa el caso de prueba. Dicho objeto contiene, entre otras cosas, un objeto heap de entrada y un objeto heap de salida, que serán usados por el `Graph Manager` para representar los heaps gráficamente.

- El `Graph Manager` es el núcleo de esta funcionalidad. Este módulo administra las representaciones gráficas basadas en el objeto heap que recibe como parámetro. Cada representación gráfica es en esencia un grafo que encapsula el heap o parte de él y contiene información sobre como debe ser mostrado por la interfaz.

  El `Graph Manager` implementa los algoritmos para la expansión y contracción de nodos así como el sistema de dibujado del grafo.

3.3.3. Dibujado automático, expansión y contracción de grafos

Dibujado automático: El dibujado automático se basa en la idea de representar el grafo como un árbol aunque no lo sea. Para ello, se distinguen dos tipos de aristas; las primarias, que formarían parte del árbol; y las secundarias, que provocan los ciclos o sirven de alias.

Para dibujar el grafo, se desactivan en primer lugar las aristas del segundo tipo. Después, se aplica un algoritmo de dibujado de árboles. Finalmente, se vuelven a activar todas las aristas y se aplica otro algoritmo que evita que las aristas se superpongan entre ellas y con los nodos.

Expansión de nodos: Dado el nodo a expandir, se obtiene una lista de las referencias que parten de éste. Para cada una de las referencias, se comprueba si el nodo destino ya existe, en cuyo caso se crea una arista secundaria. Si por el contrario el nodo no existe, se crea el nodo y se conecta con una arista primaria.

Contracción de nodos: La contracción de nodos es más compleja, ya que requiere evitar que queden nodos aislados y permitir que un nodo

cambie de padre si el padre es contraído. En líneas generales el algoritmo hace lo siguiente:

- Dado un nodo, obtiene todos sus hijos y elimina las conexiones entre padre e hijo.
- A continuación, se aplica el algoritmo de contracción a todos los hijos recursivamente.
- Una vez contraídos los hijos, se comprueba si tienen aún alguna arista entrante, si es así, la arista se convierte en primaria y el hijo cambia de padre. En caso contrario, el hijo se elimina.

Capítulo 4

Mostrando la traza de los casos de prueba

La traza de un caso de prueba es la secuencia de instrucciones que se ejercitan durante su ejecución.

Como hemos visto, nuestra herramienta hace uso de PET para la generación de casos de prueba y, de hecho, es posible visualizar la traza de un caso de prueba en PET haciendo uso del modo consola. Sin embargo, a medida que la complejidad del código aumenta, se hace más difícil leer esta información. Con el fin de facilitar la labor del programador se han desarrollado dos formas diferentes de ver las instrucciones ejercitadas por un caso de prueba generado por la herramienta.

4.1. Coloreado de la traza

En la primera modalidad, jPET puede colorear todas las lineas (y por lo tanto todas las instrucciones) pertenecientes a un caso de prueba. Para hacer esto posible se ha usado el coloreado de lineas y el uso de marcadores, ambas extensiones propias de Eclipse, que, respectivamente, se han usado para resaltar las lineas ejecutadas e indicar el número de veces y en qué orden se han ejecutado dichas lineas. Gracias a esto es posible hacerse una idea desde un punto de vista global de la traza de un caso de prueba.

Como se ha indicado, jPET usa marcadores a la izquierda de las lineas coloreadas para mostrar el *orden de ejecución*. Por ejemplo, si una linea

está marcada con un 9, significa que la linea es la novena instrucción ejecutada. Si una linea se ejecuta varias veces en un mismo caso de prueba, jPET muestra dentro del marcador todos los números que representan el orden de ejecución de dicha linea, separado por comas. Esta modalidad permite ver rápidamente si una instrucción concreta ha sido ejecutada, cuántas veces y en qué orden.

4.1.1. Usando el coloreado de la traza

Con el fin de usar esta funcionalidad, es necesario elegir primero un caso de prueba dentro de la vista de jPET y, haciendo click derecho, seleccionar la opción *Show Trace*. jPET se encarga de colorear las lineas pertenecientes al caso de prueba seleccionado. En caso de que el fichero en el que está el código fuente no esté abierto en el editor, jPET lo abrirá automáticamente.

Podemos ver cómo funciona el coloreado de la traza sobre nuestro ejemplo en la imagen 4.1. Las instrucciones ejecutadas se colorean en verde y se muestran marcadas con un símbolo a la izquierda de cada instrucción. Poniendo el ratón encima de dichos símbolos se muestra el orden de ejecución de la instrucción. En la imagen se puede ver como la instrucción seleccionada se ejecuta 2 veces, en orden 3 y 8 respectivamente.

En la imagen 4.2 podemos ver un caso especial del coloreado de la traza, el coloreado de instrucciones que generan excepción. Como se puede observar no solo se marca la instrucción concreta que genera la excepción si no que se colorean todas aquellas instrucciones que no la capturan. En nuestro ejemplo, gracias al coloreado de la traza y al resto de funcionalidades que ofrece jPET podemos detectar que hay una excepción si el array `adjacencyList` es nulo.

Como se muestra, sólo las instrucciones pertenecientes al fichero .class analizado se colorean. Esto es debido a que puede darse el caso en el que el usuario no tenga acceso al código fuente de todo el código de bytes usado.

4.1.2. Detalles de implementación

Desde el punto de vista de la implementación, hemos hecho uso de las ventajas que ofrece el entorno de Eclipse que provee una API para colorear y marcar lineas de código usando puntos de extensión tales como "annotationTypes" y "markers". Como es de esperar, Eclipse necesita saber

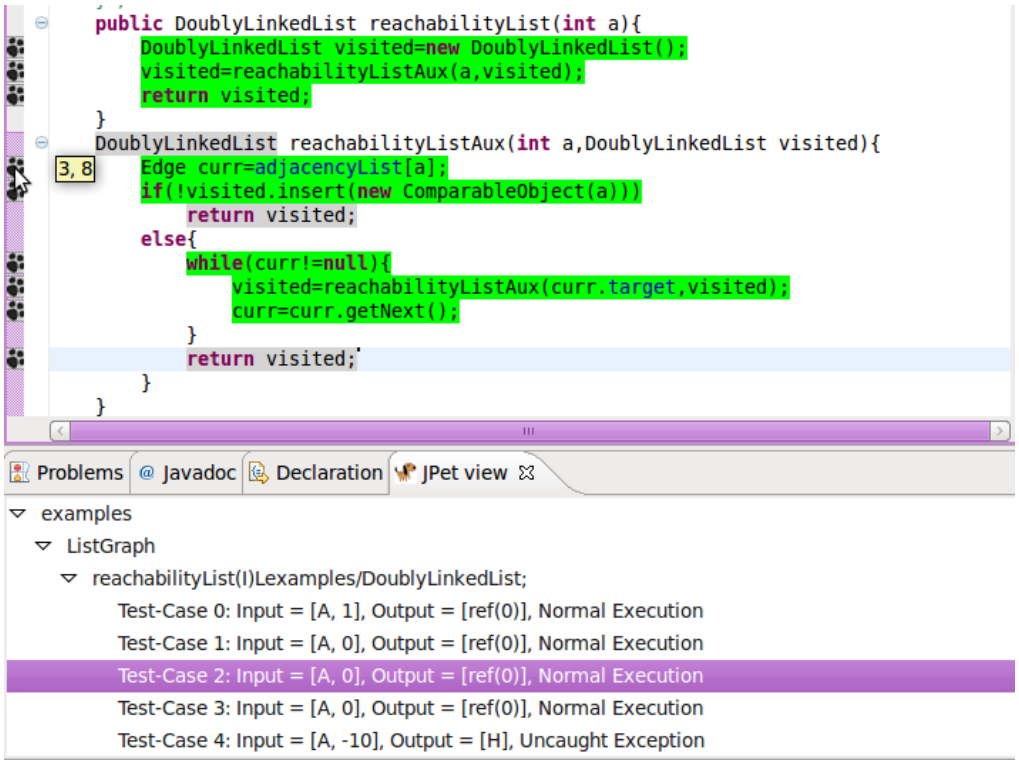

Figura 4.1: Ejemplo de coloreado de la traza normal

qué lineas de código van a ser coloreadas o marcadas. Como jPET genera los casos de prueba en base al código de bytes asociado a un programa, ha sido necesario relacionar cada instrucción de código de bytes con la linea correspondiente en el código fuente Java. Hay que tener en cuenta que cada instrucción Java puede producir varias instrucciones de código de bytes. Además, los programadores pueden escribir varias instrucciones en una misma linea. Por otro lado, cuando le enviamos los números de linea a Eclipse se hace distinción entre lineas de ejecución normal y lineas que provocan excepción, coloreando dichas lineas en verde o rojo respectivamente.

Para conseguir la correspondencia entre instrucciones de código de bytes e instrucciones del código java se incluye en el fichero XML mencionado anteriormente los índices de las instrucciones de código de bytes asociadas a cada caso de prueba. Después, haciendo uso de la librería *BCEL* [5], se parsea el fichero .class y se extrae la información necesaria en un formato

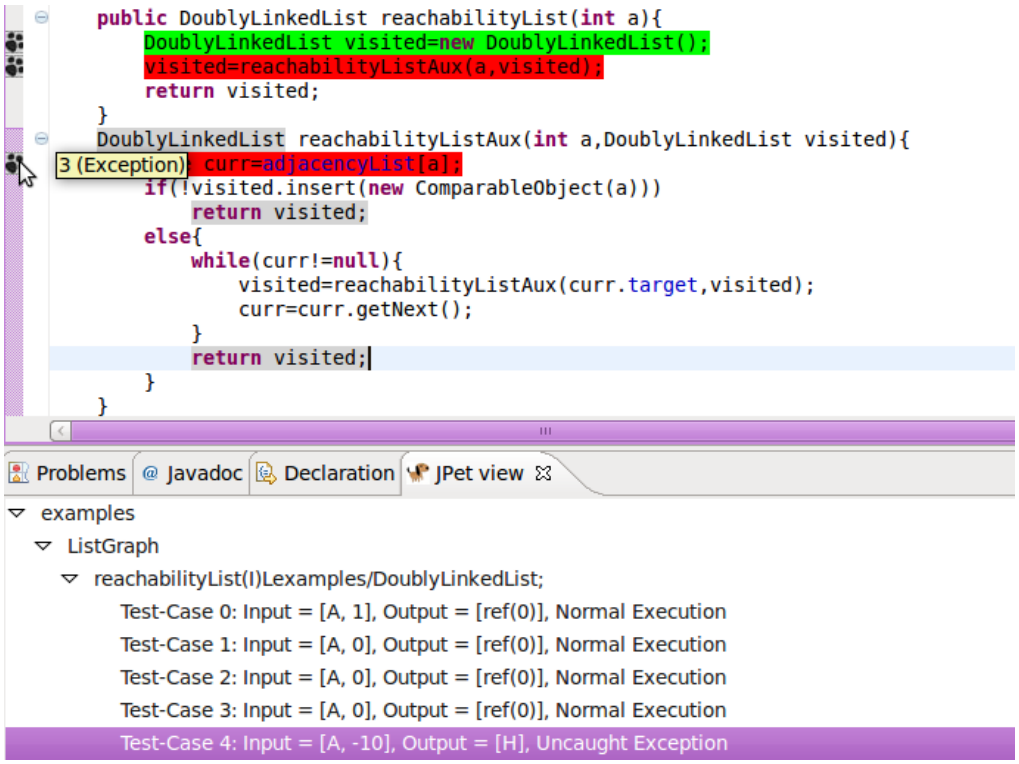

Figura 4.2: Ejemplo de coloreado de la traza con excepción

adecuado para trabajar con ella. En concreto se obtiene cada instrucción de código de bytes ejecutada y la tabla de números de linea que es la que nos permite hacer la mencionada correspondencia. La distinción entre instrucciones ejecutadas de forma normal e instrucciones que generan excepciones se hace gracias a la información proporcionada por *BCEL*.

4.2. El depurador de la traza

La segunda modalidad permite seguir la traza paso a paso desde el código fuente a través de un depurador implementado mediante las opciones de depuración propias de Eclipse. Esta modalidad necesita que la perspectiva *Debug* de Eclipse esté activa.

4.2.1. Funcionamiento

Una vez abierta la perspectiva *Debug* se puede seleccionar el caso de prueba en la jPET view y, haciendo click derecho en él, elegir la opción *Show trace debug* para iniciar el proceso de depuración del caso de prueba seleccionado. Inicialmente el depurador apunta a la primera instrucción de la traza.

Como en los depuradores estándar, jPET ofrece dos formas de avance:

- *Step over* (F6): da un paso a la siguiente instrucción sin entrar en las llamadas a métodos, en otras palabras, no salta a instrucciones que pertenezcan a otros métodos (o constructores).

- *Step into* (F5): da un paso a la siguiente instrucción de la traza, la cual puede pertenecer a un método diferente.

Supongamos que partiendo del ejemplo explicado en el capítulo 2, queremos depurar el caso de prueba 0. Hemos realizado *step over* para saltarnos la constructora de la instrucción inicial y podemos adentrarnos en el método auxiliar con *step into* o avanzar a la instrucción de *return* con *step over*. En las imágenes 4.3, 4.4, 4.5 podemos ver este proceso.

4.2.2. Encontrando errores con el depurador

El depurador de la traza nos ayuda a detectar el error localizado por el visor de casos de prueba. Siguiendo la traza mediante la opción *step into*, entramos en el método `insert` de la clase `DoubleLinkedList`, donde podemos ver que el atributo `last` de la clase, nunca se actualiza y queda apuntando siempre al primer nodo. Esto puede verse en la imagen 4.6.

4.2.3. Detalles de implementación

Paradigma de depuración de Eclipse. Para entender cómo funciona el depurador de Eclipse e implementar uno personalizado, primero debemos entender a modo de ejemplo cómo es el mecanismo para el lenguaje Java. El depurador de Eclipse se basa en el uso de dos hilos.

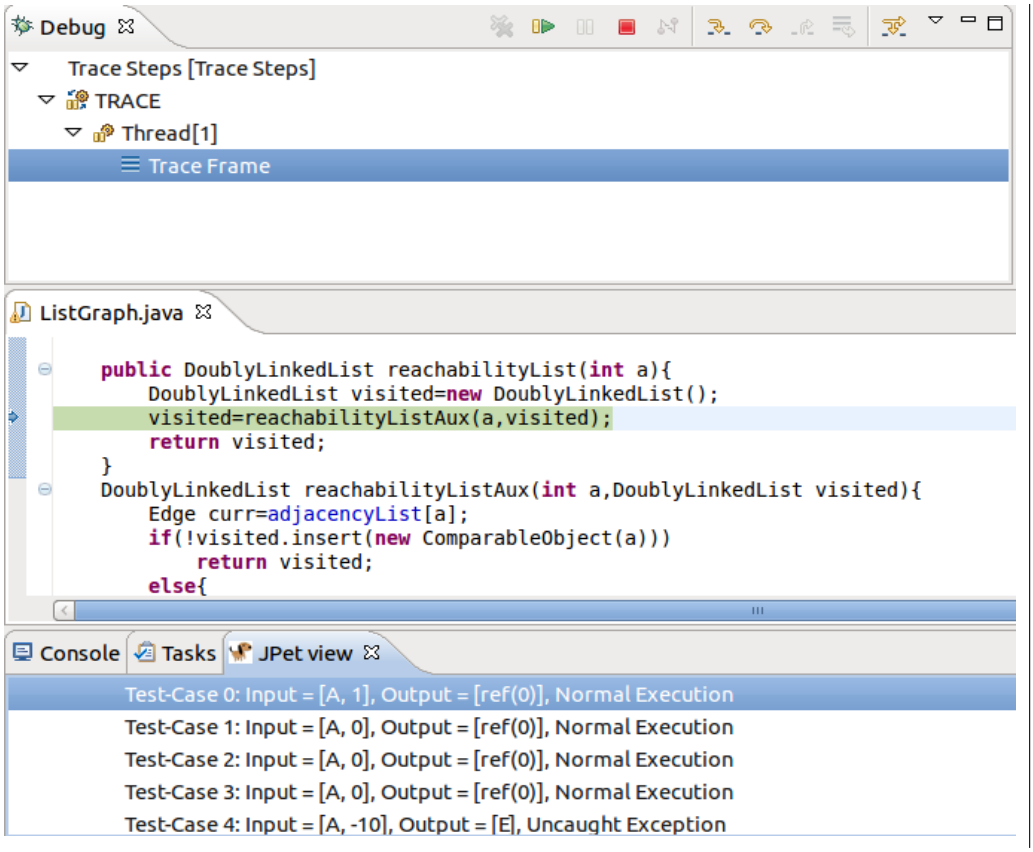

Figura 4.3: Ejemplo de depuración: Paso de partida.

Hilo de la máquina virtual de Java. Se emplea un hilo para tratar con la máquina virtual de Java. Este hilo se encarga de iniciarla, cerrarla, escuchar sus respuestas para comprobar el estado de ejecución de un programa o darle órdenes para continuar o pararlo.

Hilo controlador de la depuración. También tiene otro hilo que se encarga de controlar el proceso de depuración desde Eclipse e interpretar las órdenes que va dando el usuario. Éste manda las órdenes al hilo encargado de controlar la máquina virtual de Java. Un diagrama de secuencia de la interacción entre los hilos se puede ver en la imagen 4.7. Como puede observarse, una vez se inicia la depuración, entre ellos comienza una conversación sincronizada en la que el controlador de la depuración ordena al hilo

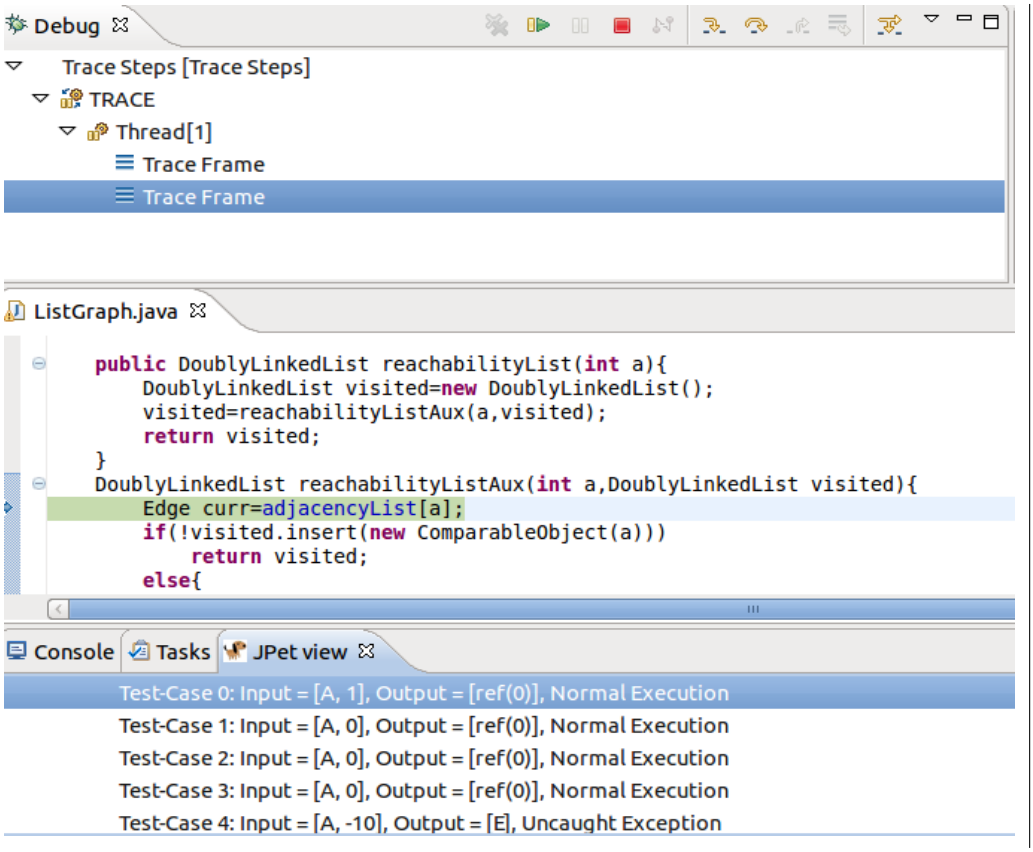

Figura 4.4: Ejemplo de depuración: Tras hacer *step into*.

de la máquina virtual de Java lo que el usuario ha pedido y espera resultados proporcionados por él. Una vez el controlador de la depuración tiene la respuesta, se encarga de mostrar al usuario el siguiente paso y realizar los pertinentes cambios en las pilas de ejecución, vistas, etc.

Debug Target. Para trasladar este paradigma de depuración a nuestro caso, hemos creado un controlador de depuración denominado *Debug Target* para controlar el proceso de depuración de la traza, y hemos usado la traza concreta del caso de prueba parseada del XML como si fuera una máquina virtual que le da las respuestas que va pidiendo el *Debug Target*.

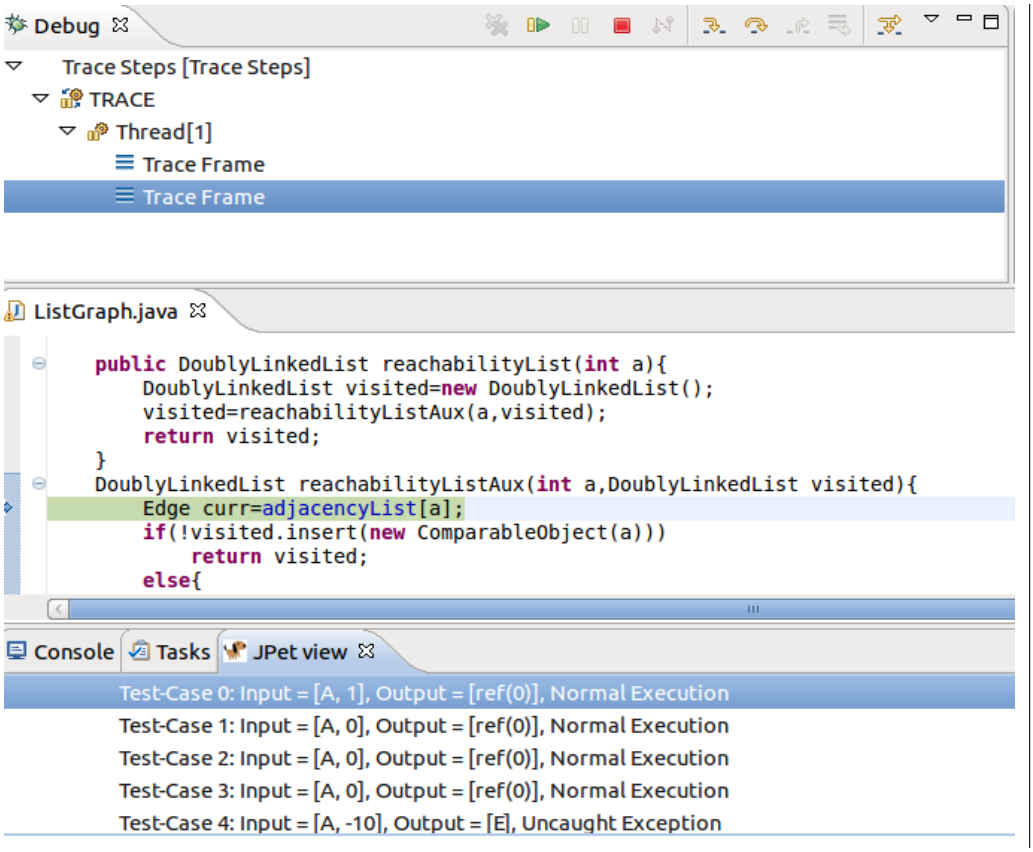

Figura 4.5: Ejemplo de depuración: Tras hacer *step over*.

Máquina virtual de la traza. A diferencia de la depuración para Java, en este caso no tenemos un programa externo en ejecución como la máquina virtual, si no que desde el propio objeto creado para almacenar la traza podemos simular su ejecución iterando en cada una de las instrucciones. De esta forma tendremos las mismas respuestas que se habrían obtenido de la máquina virtual de Java. Entre las operaciones que se le pueden ordenar a esta máquina virtual están:

- Iniciar depuración de la traza.
- Avanzar una instrucción dentro de una llamada o saltársela.
- Avanzar a la siguiente llamada.

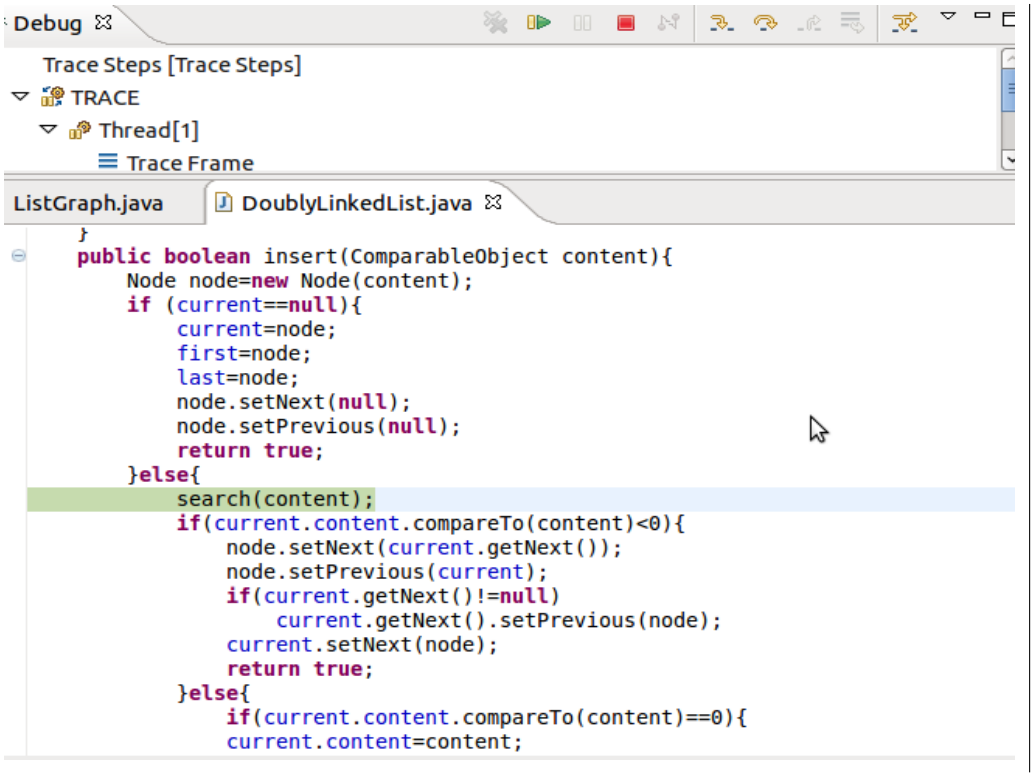

Figura 4.6: Ejemplo del error en el método insert de DoubleLinkedList.

- Obtener el número de línea en el código fuente de la instrucción actual en depuración.
- Obtener los nombres del código fuente.
- Finalizar la depuración de la traza.

Hay que tener en cuenta que la traza que ha generado PET viene en código de bytes. Para realizar todos estos pasos empleamos la asociación establecida entre código Java y código bytecode generada por el *BytecodeHandler* usando la librería BCEL como se ha explicado antes.

Controlador de la depuración. Las órdenes que da el controlador (*Debug Target*) a esta traza, tras su inicialización, dependen del usuario. Eclipse proporciona un sistema de escucha de eventos para capturar sus

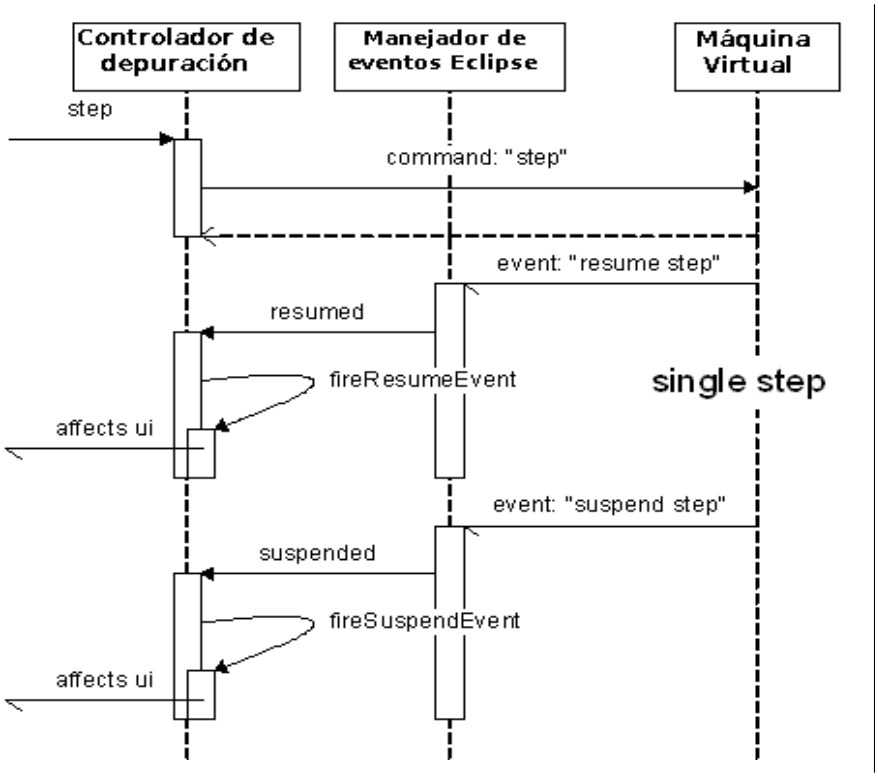

Figura 4.7: Ejemplo de comunicación entre los hilos.

peticiones, con las que el controlador de la depuración manda las órdenes necesarias a la traza para avanzar. En nuestro caso hemos empleado cinco eventos:

- *Creation*: Inicia el proceso creando los hilos y situándose al comienzo de la traza.
- *Resume*: Depura todas las instrucciones restantes de la traza y termina la depuración. Nótese que no consideramos puntos de ruptura.
- *Step over*: Avanza a la siguiente instrucción sin entrar en las llamadas a métodos.
- *Step into*: Avanza a la siguiente instrucción entrando en las llamadas a métodos.
- *Terminate*: Finaliza el proceso de depuración de la traza.

Frame Stack. En la perspectiva *Debug* de Eclipse, hay una vista que cabe destacar: la vista para la pila de llamadas. Esta vista va actualizándose a lo largo del proceso de depuración para mostrar el estado actual de llamadas a métodos en la ejecución, esta pila aumenta su tamaño cuando se hace una llamada a un método y decrece cuando se retorna.

En Eclipse, esta pila tiene otra funcionalidad adicional. Los depuradores de Eclipse apilan un objeto llamado *Frame* (de ahí el nombre de *Frame Stack*). Cada uno de estos Frames guardan la información relevante a cada llamada a método. Entre esa información está el valor de las variables, el nombre del archivo del código fuente donde está el método, la línea de depuración actual, etc.

Trace Stack Frame. Hemos implementado nuestra propia *Frame Stack* y *Frame*, de manera que contengan el número de línea actual de depuración y el nombre del fichero fuente dónde estamos depurando. La necesidad de conocer el nombre del fichero fuente es importante para poder depurar diferentes archivos en el mismo proceso de depuración. Al adentrarse en una llamada, Eclipse necesita qué archivo abrir en el editor. Por ello es necesario que la *Frame Stack* apile el *Frame* de la nueva llamada con el nombre del archivo correspondiente. De esta forma Eclipse sabe qué tiene que abrir en el editor de texto y en qué línea situar el puntero de depuración.

En la imagen 4.8 podemos ver la vista Debug en la que se puede ver el estado de esta pila con cada llamada.

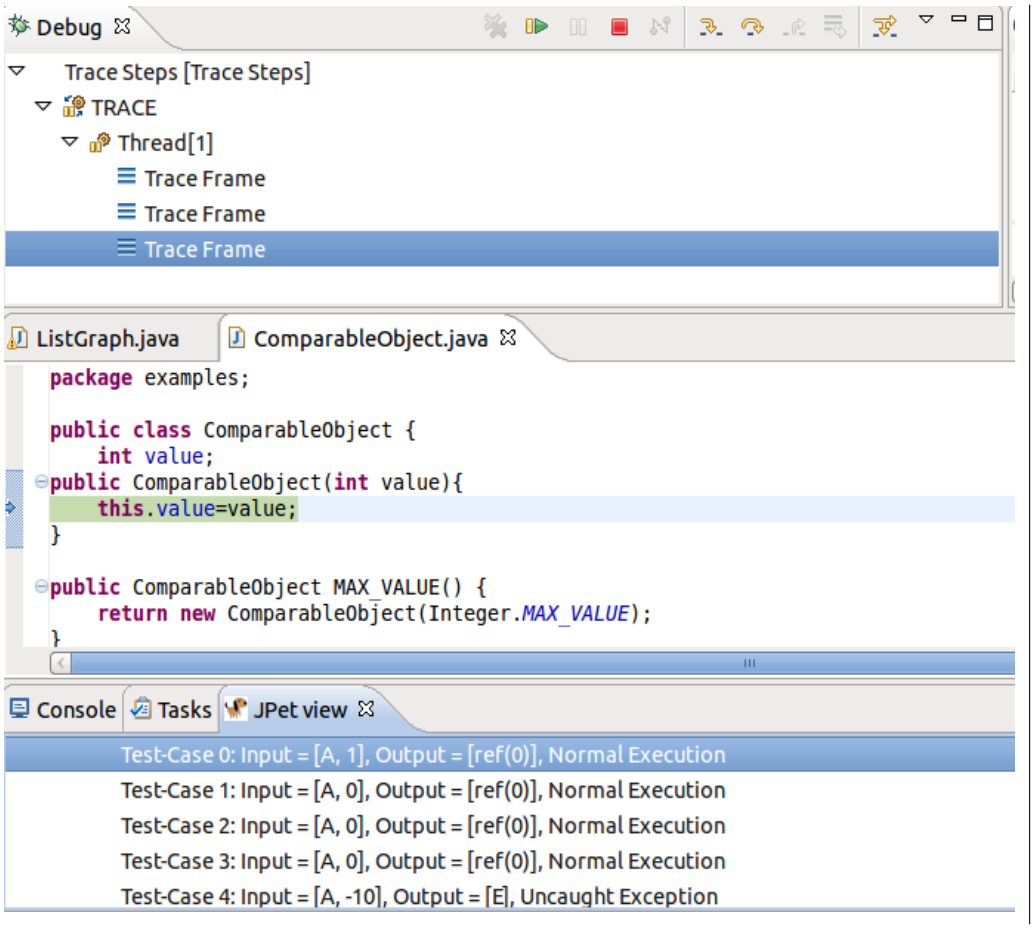

Figura 4.8: *Trace Stack Frame*: Llamada al método auxiliar.

Capítulo 5

Precondiciones en los métodos

Uno de los principales problemas de la *TDG* basada en ejecución simbólica es la gran cantidad de caminos de ejecución que se consideran, la cual crece de forma exponencial con el número de bifurcaciones del programa. Esto, por una parte plantea problemas de escalabilidad.

Por otra parte, ésto provoca la generación de un número elevado de casos de prueba, muchos de los cuales se corresponden con ejecuciones irrelevantes, por ejemplo estados incorrectos de estructuras de datos.

Una manera de aliviar esta situación es el uso de *precondiciones*. Las precondiciones permiten al desarrollador especificar restricciones en los argumentos de entrada y del objeto llamante del método o en los datos del 'heap'. De esta manera, se puede conseguir podar el árbol de ejecución simbólica que desarrolla PET y por lo tanto evitar la generación de los casos de prueba inútiles, triviales o no deseados.

5.1. Introducción a JML

Para que el desarrollador pueda escribir esas precondiciones en nuestro sistema, hemos establecido el uso de *JML*, Java Modeling Language [11], que se ha convertido en el lenguaje estándar para el software de verificación de Java.

El lenguaje *JML* ofrece la posibilidad de escribir precondiciones, postcondiciones, invariantes (reglas que se cumplen a lo largo de la ejecución) y restricciones para atributos. Sin embargo para nuestro objetivo ha sido suficiente con tomar el subconjunto de JML suficiente para poder especificar

precondiciones. Esas precondiciones se escriben antes de la cabecera de cada método que el desarrollador desea, usando la siguiente sintáxis:

```
/**@ requires precondition; */
Donde precondition es cualquier condición del lenguaje Java
como las que se usan en los bucles.
```

PET tiene un sistema de uso de precondiciones en el que descarta casos de prueba que no las cumplen usando **CLPFD**. Para usarlo hemos implementado la conexión entre el uso de JML en código fuente de Java en Eclipse con **CLPFD**.

5.2. Ejemplo de uso

Ejemplo con muchos casos. Un ejemplo del elevado número de casos de prueba que se pueden generar usando PET se puede observar subiendo a 3 el criterio de recubrimiento. Para el ejemplo explicado en el capítulo dos se generarían 82 casos de prueba en un tiempo de 48.259 ms, como se puede ver en la figura 5.1. Al añadir la precondición para que el argumento de entrada sea menor que 2 se obtienen 60 casos de prueba con un tiempo de 7.169 ms como se puede comprobar en la figura 5.2.

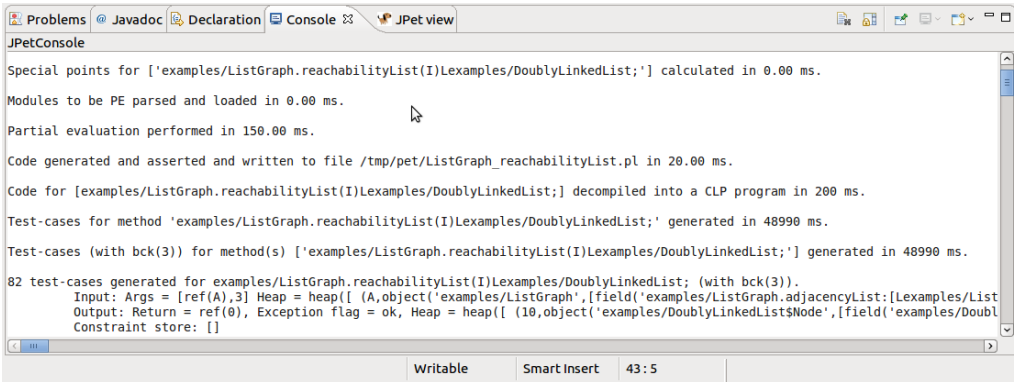

Figura 5.1: Ejemplo de resultado de PET con un elevado número de casos de prueba.

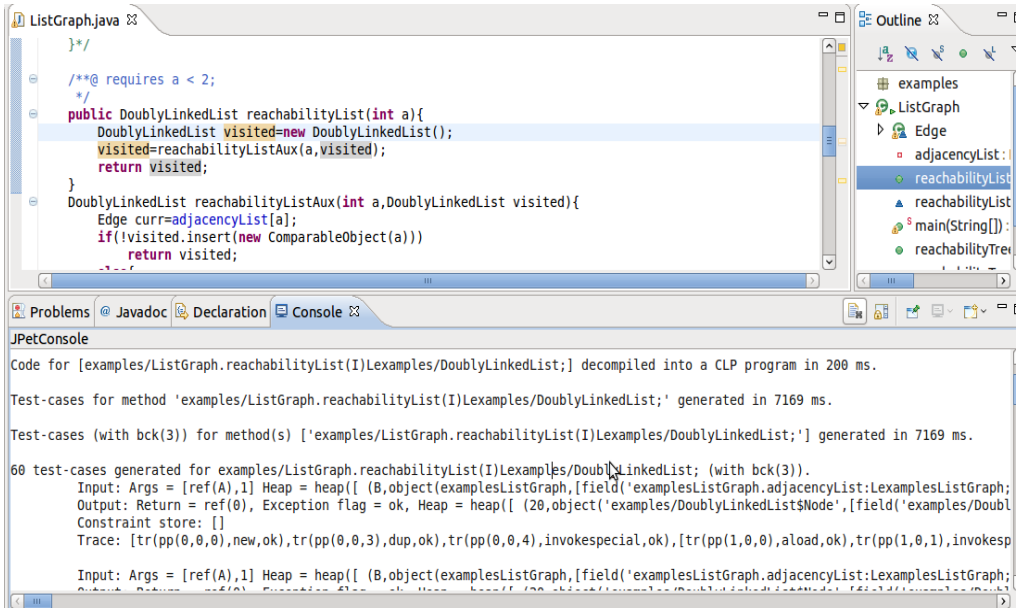

Figura 5.2: Ejemplo de resultado de PET con un menor número de casos de prueba.

Ejemplo con resultados triviales. Pero volviendo a bajar a 2 el factor de bloque, cuyo resultado son los cinco casos de prueba mencionados en el capítulo dos, podemos especificar la siguiente precondición que establece que el primer argumento del método `reachabilityList` es un número positivo y que la `adjacencyList` no es null. Esto puede ser escrito con la sintáxis de JML con la siguiente precondición:

```
/*@ requires a >= 2 ; */
public DoublyLinkedList reachabilityList(int a)
```

En la figura 5.3, podemos ver los cinco casos de prueba generados por PET sin escribir ninguna precondición, generándose casos de prueba para entradas con valor 0 (hasta donde llega el recubrimiento pedido a PET) y en la figura 5.4 se puede ver que al hacer uso de la precondición mencionada antes PET genera cinco casos de prueba más interesantes con entradas mayores.

De esta forma hemos cumplido el objetivo marcado que era quitar aquellos casos de prueba triviales o inútiles, ya que no nos interesa el caso en el

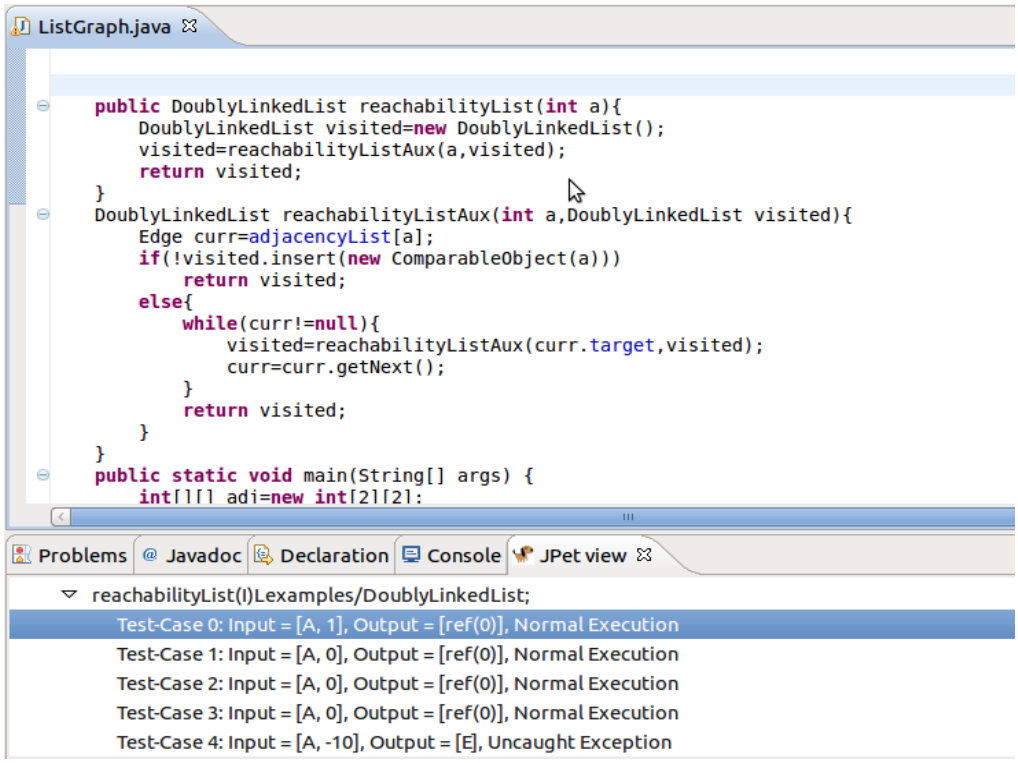

Figura 5.3: Ejemplo de resultado de PET sin precondiciones.

que la entrada para el método es cero o uno.

5.3. Detalles de implementación

Para extraer el texto de las precondiciones escritas por el desarrollador en JML, hemos empleado una utilidad proporcionada por Eclipse que extrae documentación de JavaDoc. Dicha utilidad nos permite extraer los comentarios JML, que después son interpretados utilizando un compilador generado automáticamente usando la herramienta ANTLR.

ANTLR. Hemos empleado la herramienta ANTLR para generar un compilador de los comentarios en JML. ANTLR, toma una gramática de la parte de JML que queremos capturar y nos da el correspondiente compilador en Java.

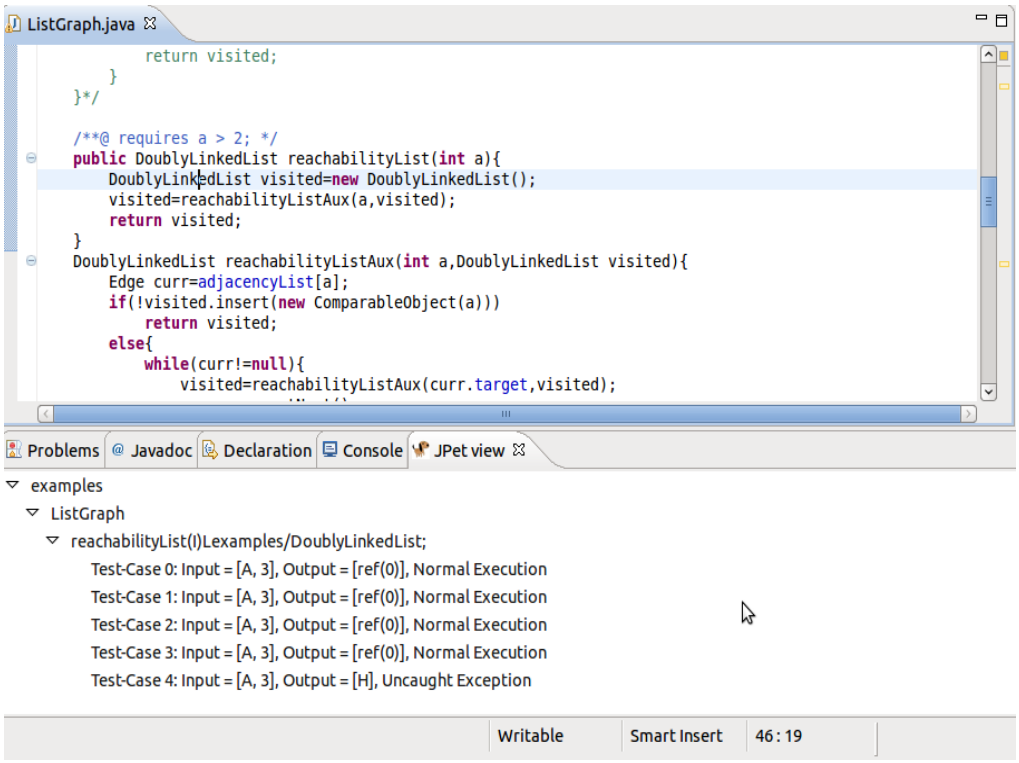

Figura 5.4: Ejemplo de resultado de PET con precondiciones.

Una de las principales características de ANTLR es que permite la inclusión de acciones semánticas entre el reconocimiento de los *tokens* de entrada. Gracias a ésto, hemos sido capaces de intercalar las acciones necesarias para generar las precondiciones en el formato que PET necesita (lista de restricciones CLP).

CLP(FD). PET requiere un término CLP [15], posiblemente utilizando predicados de la librería *clpfd* (restricciones sobre dominios finitos), escrito de la forma adecuada como formato para expresar precondiciones. Sin entrar en detalles, necesita: 1) un predicado Prolog que especifique la cabecera del método, tomando como variables sin unificar los argumentos, y 2) una lista de restricciones CLP, en las que pueden intervenir las variables de los argumentos de la parte anterior.

Ejecución en PET. Una vez tenemos las restricciones en el formato que PET necesita, éstas se añaden a la línea de comandos de la llamada a PET que generamos. De esta manera, PET las emplea para podar el árbol de ejecución simbólica y así poder reducir el número de casos de prueba generados.

Cabe destacar que las precondiciones se emplean desde el comienzo de la búsqueda, lo que permite podar las ramas de ejecución tan pronto como son infringidas, reduciéndose así el tiempo de generación de casos de prueba.

Capítulo 6

Conclusiones y trabajo futuro

En el presente proyecto se ha desarrollado jPET, una herramienta integrada en Eclipse que genera automáticamente casos de prueba para programas Java. jPET dispone de funcionalidades que facilitan al usuario el aprovechamiento de los casos de prueba generados. Es por esto que consideramos que cumple los requisitos inicialmente planteados, es decir, proporcionar una importante ayuda a los programadores durante la fase de pruebas en los procesos de desarrollo de software.

Cada una de las funcionalidades de jPET anteriormente explicadas se corresponden con los distintos requisitos de una herramienta de este tipo:

- La completa integración de la herramienta en el entorno de desarrollo Eclipse y la gestión de las preferencias internas de PET, permiten un uso sencillo, rápido y práctico de la herramienta. Ésto hace factible su integración en los procesos de desarrollo de software con un coste de aprendizaje mínimo.

- El visor de casos de prueba ayuda al usuario a comprender la información generada por jPET. De esta forma, el usuario puede comprender mejor el comportamiento del código probado y su correspondencia con el comportamiento esperado. En conclusión, el visor maximiza las posibilidades del usuario de encontrar fallos en el programa en desarrollo.

- El coloreado y depuración de la traza, por otro lado, permiten estrechar la conexión entre los casos de prueba generados y el código

fuente de donde provienen. Ésto ayuda a localizar los fallos de programación causantes del mal funcionamiento de la aplicación.

- La especificación de precondiciones permite un filtrado de los casos de prueba generados. Ésto proporciona mejoras en cuanto a la escalabilidad y facilidad de uso del sistema, permitiendo su aplicación a códigos más complejos y paliando en parte la explosión combinatoria en el número de casos de prueba generados. Además, esta característica no presenta un gran aumento de dificultad desde el punto de vista del uso de la aplicación, al valerse de una notación familiar para el programador.

Este proyecto ha supuesto una serie de desafíos y dificultades variadas. Se ha trabajado en todo momento sobre la arquitectura de plugins de Eclipse, por lo que toda decisión de diseño ha tenido que adaptarse a los procedimientos especificados por dicha arquitectura. Nos hemos enfrentado a multitud de problemas distintos que se resumen a continuación:

- La ejecución de PET ha requerido de una gestión básica de procesos y de sus entradas y salidas. Además, hemos aprendido sobre la gestión de la concurrencia en una herramienta de gran envergadura como es Eclipse.

- El desarrollo del visor de casos de prueba nos ha permitido trabajar en interfaces gráficas avanzadas y hacer uso de sistemas de dibujado de grafos. Hemos obtenido experiencia práctica para afrontar el desarrollo de interfaces y sistemas centrados en la interacción con el usuario.

- La comunicación con PET ha requerido la generación y tratamiento de archivos XML, un estándar cuyo uso está enormemente extendido.

- Tanto para el coloreado como para la depuración de trazas ha sido necesario acceder y tratar archivos binarios de bytecode (formato .class).

- Para el procesamiento de precondiciones se ha desarrollado un analizador sintáctico. Para ésto se han aplicado conceptos de procesadores de lenguaje y se ha hecho uso de herramientas de uso profesional

(ANTLR). Además, las precondiciones se traducen al formato que admite PET, basado en CLP . Esto ha requerido aprender acerca del funcionamiento y sintaxis de las expresiones CLP.

- Por último, la creación de jPET ha requerido un conocimiento avanzado del sistema PET y su motor de generación de casos de prueba por medio de ejecución simbólica en CLP.

A medida que ha avanzado el desarrollo de jPET se han podido detectar una serie de funcionalidades o posibles expansiones que mejorarían la herramienta y aumentarían su utilidad. Estas funcionalidades no se han abordado en el presente proyecto y pueden abrir paso a nuevas investigaciones y líneas de trabajo. A continuación se describen dichas mejoras:

- Como ya se sabe, PET se encuentra en el núcleo de jPET por lo que cualquier mejora de PET mejorará consecuentemente el funcionamiento de jPET. En concreto, PET podría expandirse para usar ejecución concólica u otra alternativa, que permita superar las limitaciones de la ejecución simbólica pura. Estas limitaciones son, la incapacidad de trabajar con librerías de código nativo, llamadas al sistema operativo o interacción avanzada con el usuario.

- El sistema de especificación de precondiciones supone un incremento importante en la escalabilidad de la herramienta. Esta nueva funcionalidad podría ser mejorada significativamente ampliando el sistema de precondiciones para soportar expresiones más complejas.

- El proceso de testing podría automatizarse en mayor medida evitando que el usuario tenga que hacer el papel de oráculo manualmente y caso por caso. El reconocimiento de postcondiciones supondría un gran avance en este sentido. Las postcondiciones son propiedades que han de cumplirse al final de la ejecución de un código y definen su comportamiento. El sistema podría comprobar el cumplimiento de éstas automáticamente y alertar al usuario en el caso de que no se cumplan.

- En esta misma linea, otra posibilidad puede ser permitir la especificación de *invariantes de clase*, propiedades de los atributos que deben

cumplirse al inicio y final de todos los métodos públicos. La especificación de invariantes de clase permitiría completar la información proporcionada en las precondiciones, reduciendo en gran medida el número de casos de prueba no relevantes. Además, al igual que las postcondiciones, permitiría la detección directa de errores al poder comprobar si los invariantes se cumplen al final de la ejecución de los métodos.

- El visor de casos de prueba es una herramienta centrada en la interacción con el usuario. Éste podría mejorarse incrementando su flexibilidad y permitiendo al usuario configurarlo. El usuario podría, en ese caso, adaptar la herramienta a sus gustos y necesidades pudiendo aumentar su rendimiento y mejorando su experiencia de uso.

- Sería interesante ampliar el visualizador de trazas para que incluya el coloreado de la traza de ficheros fuente secundarios siempre que éstos estén disponibles. El sistema actual solo colorea las líneas pertenecientes al fichero fuente analizado ya que cualquier otro puede no estar disponible.

- La posibilidad de usar puntos de ruptura en el depurador de trazas supondría un añadido interesante. Gracias a éstos, el usuario podría agilizar la depuración examinando la parte del código que le interese de forma directa. Esta característica puede ser crítica en casos donde la traza sea muy extensa y el avance paso a paso no sea práctico.

Bibliografía

[1] E. Albert, M. Gómez-Zamalloa, and G. Puebla. PET: A Partial Evaluation-based Test Case Generation Tool for Java Bytecode. In *Proc. of. PEPM'10*. ACM Press, 2010.

[2] Elvira Albert, Israel Cabañas, Antonio Flores-Montoya, Miguel G-Zamalloa, and Sergio Gutiérrez. Software Testing using jPET. http://costa.ls.fi.upm.es/pet/papers/AlbertCFGG11.pdf, 2011.

[3] David Benson and Gaudenz Alder. JGraphX (JGraph 6) User Manual. http://www.jgraph.com/doc/mxgraph/index_javavis.html.

[4] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.

[5] Markus Dahm, Jason van Zyl, Enver Haase, Dave Brosius, and Torsten Curdt. BCEL. http://jakarta.apache.org/bcel/manual.html, 2006.

[6] C. Engel and R. Hähnle. Generating unit tests from formal proofs. In *Proceedings of TAP*, volume 4454 of *LNCS*. Springer, 2007.

[7] M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test Case Generation for Object-Oriented Imperative Languages in CLP. *TPLP, ICLP'10 Special Issue*, 2010.

[8] Gordon Hirsch. Swing/SWT Integration. http://www.eclipse.org/articles/article.php?file=Article-Swing-SWT-Integration/index.html, June 2007.

[9] Cem Kaner, James Batch, and Bret Pettichord. *Lessons Learned in Software Testing: A Context Driven Approach*. Microsoft Press, 2002.

[10] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[11] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. JML Reference Manual. http://www.eecs.ucf.edu/~leavens/JML//jmlrefman/jmlrefman_toc.html, September 2009.

[12] Koushik Sen and Gul Agha. Cute and jcute : Concolic unit testing and explicit path model-checking tools. In *CAV*, 2006.

[13] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *SIGSOFT*, 2005.

[14] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .NET. In *TAP*, pages 134–153, 2008.

[15] Markus Triska. CLP(FD). http://www.swi-prolog.org/man/clpfd.html, 2011.

Apéndice A

Especificación del fichero XML

```
[XML] ::= <pet>
            [TestCase]*
          </pet>

[TestCase] ::= <test_case>
                 [Firma]
                 [ArgsIn]
                 [HeapIn]
                 [HeapOut]
                 [Return]
                 [ExcFlag]
                 [Trace]
               </test_case>

[Firma] ::= <method>
              'Firma_del_metodo'
            </method>

[ArgsIn] ::= <args_in>
               ([Ref]|[Data])*
             </args_in>

[HeapIn] ::= <heap_in>
               [Elem]*
             </heap_in>
```

```
[HeapOut] ::= <heap_out>
                [Elem]*
              </heap_out>

[Return] ::= <return>
               'Valor de retorno'
             </return>

[ExcFlag] ::= <exception_flag>
                'Flag de excepcion'
              </exception_flag>

[Trace] ::= <trace>
              [Call]*
            </trace>

[Ref]  ::= <ref>
             'Nombre del parametro'
           </ref>

[Data] ::= <data>
             'Valor del parametro'
           </data>

[Elem] ::= <elem>
             [Num]
             ([Obj]|[Array])
           </elem>
[Num]  ::= <num>
             'Nombre del elemento'
           </num>

[Obj]  ::= <object>
             <class_name>
               'Nombre de la clase'
             </class_name>
             [Fields]
```

```
            </object>
[Fields]  ::= <fields>
                [Field]*
              <fields>
[Field]  ::= <field>
               <field_name>
                 'Nombre del campo'
               </field_name>
               ([Ref]|[Data])*
             </field>
[Array]  ::= <array>
               <type>
                 'Tipo del array'
               </type>
               <num_elems>
                 'Numero de elementos'
               </num_elems>
               [Args]
             </array>
[Args]  ::= <args>
              [Ref]
              [Arg]*
            </args>
[Arg]  ::= <arg>
             'Nombre o valor de la variable'
           </arg>

[Call]  ::= <call>
              [Depth]
              [ClassName]
              [MethodName]
              [Instruction]*
            </call>
[Depth]  ::= <depth>
               'Profundidad de la llamada'
             </depth>
[ClassName]  ::= <class_name>
                   'Clase en la que se esta ejecutando'
```

```
                    </class_name>
[MethodName]  ::= <method_name>
                        'Firma del metodo en el que se esta ejecutando'
                    </method_name>
[Instruction]  ::= <instruction>
                        [PC]
                        [Bytecode]
                    </instruction>
[PC]  ::= <pc>
              'Contador de programa para la instruccion de codigo de bytes'
          </pc>
[Bytecode]  ::= <bytecode_name>
                      'Nombre de la instruccion de codigo de bytes'
                  <bytecode_name>
```